FACULTAD DE CIENCIAS

UNIVERSIDAD DE CANTABRIA Facultad de Ciencias

Proyecto Fin de Carrera

DISEÑO E IMPLEMENTACION DE UN SISTEMA DE GESTION DE LOG PARA SU INTEGRACION EN UN SOFTWARE DE MENSAJERIA DE BAJA LATENCIA

(DESIGN AND IMPLEMENTATION OF A LOG MANAGEMENT SYSTEM FOR ITS INTEGRATION INTO A LOW LATENCY MESSAGING SOFTWARE)

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Sergio Gorrochategui Blanco

Febrero - 2013

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Sergio Gorrochategui Blanco
Director del PFC: Roberto Juárez Maldonado
Ponente: Inés González Rodríguez

Título: "DISEÑO E IMPLEMENTACION DE UN SISTEMA DE GESTION DE LOG PARA SU INTEGRACION EN UN SOFTWARE DE MENSAJERIA DE BAJA LATENCIA"

Title: "DESIGN AND IMPLEMENTATION OF A LOG MANAGEMENT SYSTEM FOR INTEGRATION INTO A LOW LATENCY MESSAGING SOFTWARE "

Presentado a examen el día:

para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:
Presidente: Michael González Harbour
Secretaria: María del Carmen Martínez Fernández
Vocal: Rafael Menéndez de Llano Rozas
Vocal: Pablo Sánchez Barreiro
Vocal: Roberto Sanz Gil

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: Vocal	Fdo.: Vocal
Fdo.: Vocal	Fdo.: El Director del PFC

Resumen

Este proyecto de fin de carrera se ha desarrollado en la empresa Comunytek S.L., que se encarga de diseñar y comercializar sistemas de trading. Un sistema de trading analiza el mercado, comprueba las cotizaciones de los productos en tiempo real y cuando se cumplen las condiciones que marca su estrategia, genera una orden de compra o de venta, que se transmite al mercado. Todo este sistema funciona bajo una compleja arquitectura de envío de mensajes (órdenes, confirmaciones, rechazos…) entre varios emisores (clientes o terminales) y receptores (mercados). Cuanto más rápido sea un sistema de trading, antes dispondrá de la información y dará mayor ventaja a sus clientes.

Una de las líneas de trabajo que tiene la empresa es implementar un nuevo sistema de trading de baja latencia. El objetivo global es crear un sistema que sea capaz de soportar altas tasas de mensajes y que el coste de cada operación sea mínimo para garantizar la actualización y procesamiento de la información en el menor tiempo posible.

El desarrollo del nuevo sistema de trading está dividido en varias partes y en concreto, este proyecto fin de carrera se centrará en la parte encargada de integrar un sistema de log que gestionará los eventos que se produzcan dentro de una aplicación cualquiera que utilice el sistema de trading. El objetivo principal del proyecto fin de carrera será que el sistema de log sea tan rápido y eficiente como el sistema de trading en cualquier aspecto de rendimiento, lo que significa que el cómputo global de todas sus operaciones no puede influir negativamente en el rendimiento general de todo el sistema.

Para alcanzar este objetivo, se tendrán que realizar las siguientes tareas:

- Analizar los actuales sistemas de log para establecer una arquitectura base del sistema.
- Diseñar e implementar un sistema de log que se ajuste a los requisitos de rendimiento generales del sistema.
- Integrar todo el sistema de log en el sistema de mensajería de baja latencia para su posterior utilización en una aplicación de trading.

Abstract

This dissertation has been developed in the company Comunytek S.L, which is responsible for designing and marketing trading systems. A trading system analyzes the market, checks the prices of the products in real time and when the conditions which mark its strategy hold, generates a purchase or sales order, which is transmitted to the market. The whole system operates under a complex messaging architecture (orders, confirmations, rejections etc.) among several issuers (clients or terminals) and receivers (markets). The faster a trading system is, the sooner the information will be available and the greater will be the benefit provided to its customers, implying increased revenue for both parties.

One of the company's lines of work is implementing a new system for low-latency trading. The overall objective is to create a system that is capable of supporting high message rates and such that the cost of each operation is the lowest possible consumption to ensure that the updating and processing of information take place in the shortest possible time.

The development of the new trading system is divided into several parts, and specifically, this dissertation will focus on the part in charge of integrating a log system that will manage all events that occur within an application using the trading system. The main objective of the project will be that the log system be as fast and efficient as the trading system in every aspect of performance, which means that the aggregate of all its operations cannot adversely affect the overall performance of the whole system.

To achieve this goal, we will need to perform the following tasks:

- Analyze existing log systems to establish a basis for the system architecture.
- Design and implement a logging system that meets the requirements of overall system performance, improving the current log system.
- Integrate all system logs in the system of low-latency messaging for later use in a trading application.

Índice

Índice de figuras **7**
Índice de tablas **8**
Capítulo 1. Introducción **9**
1.1 Contexto de trabajo 9
1.2 Presentación del problema. Motivación. 9
1.3 Objetivos detallados 11
1.4 Definiciones 12
Capítulo 2. Análisis de situación **13**
2.1 Sistema de Gestión de Log. 13
2.1.1 Loggers. 13
2.1.2 Appenders. 14
2.1.3 Layouts. 14
2.2 Análisis de Requisitos y Funcionalidades 15
2.3 Alternativas Actuales 15
2.3.1 Log4j 16
2.3.2 Logback 16
2.3.3 Logback vs Log4fj 16
2.3.4 Appenders de Logback 17
2.4 Test de Rendimiento para el LLFB y los Appenders de Logback 17
2.4.1 Test de rendimiento del LLFB 17
2.4.2 Test de rendimiento del sistema UM utilizando Appenders de Logback 19
2.4.3 Resultados de los test 23
Capítulo 3. Diseño y modelo de desarrollo **25**
3.1 Modelo evolutivo en espiral 25
3.2 Descripción de requisitos 26
3.3 Esquema de desarrollo 27
Capítulo 4. Implementación de la solución propuesta **28**
4.1 Modelo de programación 29
4.2 Modelo de eventos de log 30
4.2 Appender Híbrido: HybridFileAppender 31
4.2.1 LogEvent 31
4.2.2 Implementación 32
4.2.3 Tiempos de de ejecución de las operaciones (latencias) 34
4.2.4 Modos de funcionamiento 35
4.3 Appender Ultra Messaging: UMAppender 36
4.3.1 Low Latency Framework Bus 37
4.3.2 LogEventUM 39
4.3.2.1 Serialización de datos 39
4.3.3 Implementación 40
4.3.4 Modos de uso 42
Capítulo 5. Pruebas **44**
5.1 Pruebas de Serialización 44
5.1.1 Protocol Buffer vs SerializaciónUM 45
5.1.2 Conclusiones 48
5.2 Pruebas para los Appenders 48
5.2.1 HybridFileAppender 49
5.2.2 UMAppender 50
5.2.3 Appender Logback 50
5.2.4 Comparativa de resultados 50
5.2.5Análisis de los resultados 53
Capítulo 6. Integración en un software de mensajería instantánea: Trading Bank. **55**
6.1 Integración del appender en el Low Latency Framework Bus 55

6.2 Uso del LLFB y los Appenders: Aplicaciones de Trading Bank 56
Capítulo 7. Conclusiones. 58
Capítulo 8. Líneas futuras de trabajo 59
Bibliografía 61
Anexo: Contenido del CD 63

Índice de figuras

Figura 1 - Esquema de la plataforma Plug'n Trade 10
Figura 2 - Nivel jerárquico de comportamiento de los loggers 14
Figura 3 - Resumen del rendimiento del LLFB para 1000 mensajes con los dos métodos de envío 18
Figura 4 - Resumen del rendimiento del appender ConsoleAppender utilizado en el LLFB para distinta cantidad de mensajes con los dos métodos de envío considerados. 20
Figura 5 - Resumen del rendimiento del appender FileReaderAppender utilizado en el LLFB para mil mensajes con los dos métodos de envío considerados. 21
Figura 6 - Resumen del rendimiento del appender AsynAppender utilizado en el LLFB para mil mensajes con los dos métodos de envío considerados. 22
Figura 7 - Resumen del rendimiento de los tres appenders utilizando en el LLFB para 1000 mensajes con el método de envío considerado anteriormente Flush. 22
Figura 8 - Resumen del rendimiento de los tres appenders utilizados anteriormente con el LLFB para 1000 mensajes con el método de envío considerado anteriormente No-Flush.. 23
Figura 9 - Diagrama de clases de de los appenders de Logback.. 28
Figura 10 - Esquema visual del Low latency Framework Bus. 38
Figura 11 - Vista de los percentiles enviando mensaje por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM. 45
Figura 12 - Vista de latencia mínima, máxima y media enviando mensaje por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM. 46
Figura 13 - Vista del Throughput enviando 10.000 mensajes por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM.. 47
Figura 14 - Vista del Throughput enviando 100.000 mensajes por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM.. 48
Figura 15 - Comparativa de los appenders usando el modo de captura del contenido del mensajes LogBackLayout.. 51
Figura 16 - Comparativa de los appenders usando el modo de captura de contenido del mensaje Only_Message. 51
Figura 17 - Comparativa de los Appenders usando el modo de captura de contenido del mensaje *AllComplete* 52
Figura 18 - Comparativa de los Appenders desarrollados y el mejor appender de Logback (AsynAppender) 52

Índice de tablas

Tabla 1 - Rendimiento del LLFB enviando mensajes en modo Flush. 18
Tabla 2 - Rendimiento del LLFB enviando mensajes en modo No-Flush 18
Tabla 3 - Rendimiento del LLFB utilizando el appender ConsoleAppender enviado los mensajes con el modo Flush. 19
Tabla 4 - Rendimiento del LLFB utilizando el appender ConsoleAppender enviado los mensajes con el modo No-Flush. 19
Tabla 5 - Rendimiento del LLFB utilizando el appender FileReaderAppender enviado los mensajes con el modo Flush. 20
Tabla 6 - Rendimiento del LLFB utilizando el appender FileReaderAppender enviado los mensajes con el modo Flush. 20
Tabla 7 - Rendimiento del LLFB utilizando el appender AsynAppender enviado los mensajes con el modo Flush. 21
Tabla 8 - Rendimiento del LLFB utilizando el appender AsynAppender enviado los mensajes con el modo No-Flush. 21
Tabla 9 - Resultados de los percentiles para Protocol Buffers y SerializaciónUM. 45
Tabla 10 - Resultados estadísticos en términos de latencia para la serialización de datos enviando mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM. 46
Tabla 11 - Resultados estadísticos en términos de Throughput para la serialización de datos enviando 10.000 mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM. 47
Tabla 12 - Resultados estadísticos en términos de Throughput para la serialización de datos enviando 10.000 mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM. 48
Tabla13 - Resultados de todos los datos estadísticos en términos de latencia del HybridFileAppender. 49
Tabla 14 - Resultados de todos los datos estadísticos en términos de latencia del UMAppender. 50
Tabla 15 - Resultados de todos los datos estadísticos en términos de latencia del FileReaderAppnder de Logback y el AsynAppender.. 50

Capítulo 1. Introducción

A lo largo de ésta memoria, se explicará y desglosará parte del trabajo realizado durante un periodo de 4 meses en la empresa Comunytek Consultores S.L. Debido a que el trabajo realizado a tiempo completo en la empresa excede con mucho la extensión de un PFC, de entre las diversas tareas realizadas, se ha optado por desarrollar la que se ha encargado de diseñar e implementar un sistema de gestión de log para una aplicación de trading financiero.

Esta tarea es la que ha exigido mayor tiempo y dedicación y por la cantidad de trabajo realizado, es equiparable a un PFC.

1.1 Contexto de trabajo

Comunytek es una compañía consultora en tecnologías de la información y las comunicaciones con experiencia en la implantación de tecnologías avanzadas en entornos multidisciplinares. Cuenta con un equipo de más de 40 profesionales con fuerte experiencia en mercados de capitales y financieros.

Actualmente, la compañía desarrolla productos especializados para solucionar problemáticas muy concretas, como plataformas software que facilitan la rápida puesta en marcha de aplicativos o soluciones integradas con sistemas existentes.

El producto más significativo e importante que desarrolla la empresa se llama "*Plug'n Trade*" [1]: una suite modular de productos financieros que facilitan la conectividad con los mercados, la gestión de órdenes, la liquidación y la integración con plataformas corporativas de *middle* y *back-office* [2].

A día de hoy, la empresa tiene completamente desarrollada una versión estable del producto. Pero, debido a que la tecnología, las necesidades y los requisitos cambian, la compañía desea elaborar una nueva versión del software *Plug'n Trade* adaptándolo a las nuevas situaciones y circunstancias.

El propósito general de todo el proyecto de la empresa será desarrollar una nueva versión del producto más versátil y eficiente, mejorando muchos de los aspectos, especialmente en rendimiento, de la versión anterior. Dado que es un proyecto muy amplio y ambicioso, se ha dividido todo el trabajo en diferentes áreas las cuales tienen asignadas diferentes tareas para llevar a cabo.

Esta memoria se desarrolla en el área correspondiente al diseño e implementación del gestor de log de la aplicación. Se expondrán como han sido los procedimientos de diseño, creación, implementación, pruebas y resolución de problemas que se han tenido que realizar para completar los objetivos y tareas pertenecientes a esta área.

1.2 Presentación del problema. Motivación.

La funcionalidad básica de un software financiero es el envío de órdenes desde un terminal hacia un mercado financiero recibiendo una determinada respuesta. El envío de estas órdenes se hace a través de mensajes, teniendo que viajar por distintos módulos y conectores intermedios.

La principal característica de un buen software de trading financiero es que sea capaz de enviar y recibir el mayor número de mensajes por segundo, consiguiendo actualizaciones de los estados de los mercados y de las órdenes en tiempo real, siendo capaz de ejecutar cualquier operación en el menor tiempo posible.

Esta característica es la más importante porque para un usuario, el poder disponer de las actualizaciones de los mercados antes que otros usuarios, le dota de mayor ventaja competitiva a la hora de poder realizar una determinada operación y por lo tanto, poder seleccionar y conseguir las mejores ofertas del mercado.

Por lo tanto, el objetivo principal de un software financiero es ser capaz de enviar y recibir un mensaje de un extremo a otro en el menor tiempo posible, antes que cualquier otro usuario dentro del mercado.

En la figura 1, se muestra un pequeño esquema de cuáles son los módulos y conectores intermedios de la plataforma Plug'n Trade que gestionan el contenido y el camino por el que debe viajar un mensaje.

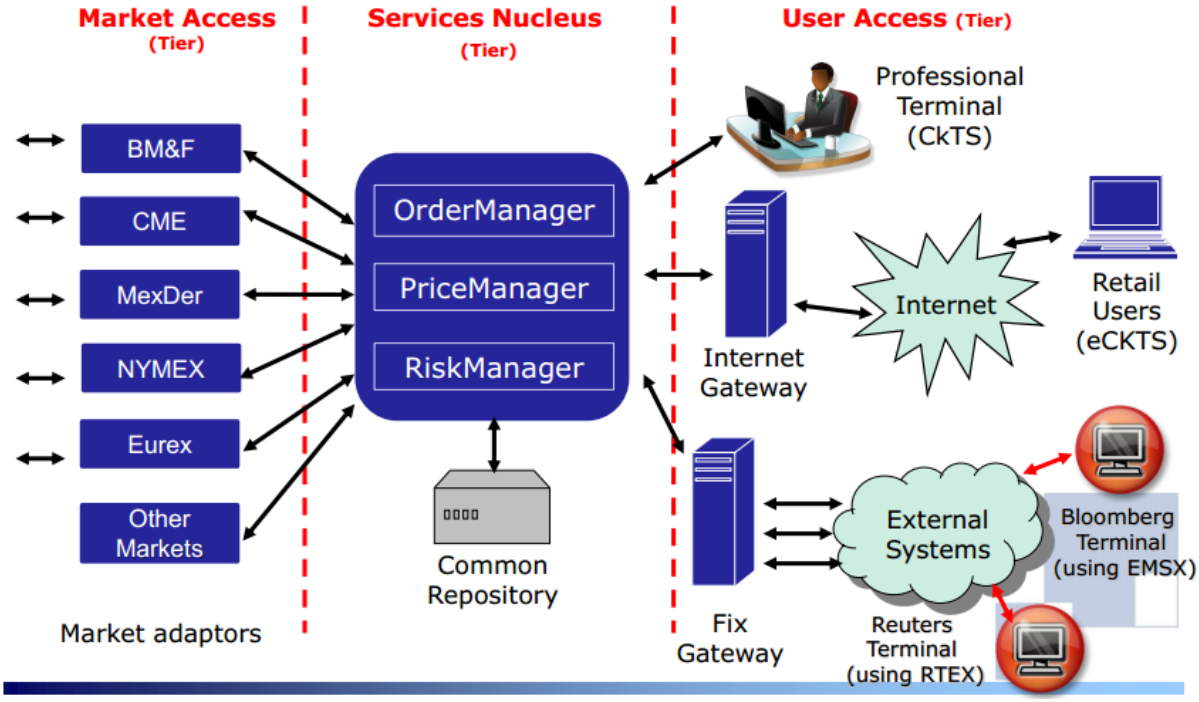


Figura1: Esquema de la plataforma Plug'n Trade, *"fuente: Comunytek"*

Los ***User Access*** son la parte del sistema donde se generan los mensajes. Existen varias formas de generar los mensajes (a través de terminales fijas, terminales controladas por los usuarios etc.) y todos ellos viajan a las puertas de enlace que conectan con el núcleo del sistema.

El ***Services Nucleus*** es la parte del sistema que se encarga de analizar, codificar y enviar los mensajes a su destino correspondiente. Los mensajes pueden ser bidireccionales y dependiendo del tipo de mensaje, puede tener que pasar por diferentes filtros para realizar una función determinada.

El ***Market Access*** es la parte del sistema que recibe los mensajes enviados por los diferentes usuarios. Cada mercado habla un lenguaje diferente de codificación, por lo que los mensajes han sido previamente codificados por el *Services Nucleus* al lenguaje que corresponda según el mercado. Los mensajes pueden ser aceptados o rechazados y habitualmente generan una respuesta de vuelta al usuario.

El principal problema que se puede encontrar un mensaje para poder ser enviado rápidamente es la latencia adicional que añade cada módulo hasta que llega a su destino, por lo que cada módulo ha de diseñarse eficazmente procurando que añada la mínima latencia al mensaje, tanto en su envío como en su recepción.

Todos los conectores o módulos han de ser muy rápidos. El retraso de solo uno de ellos puede entorpecer el buen trabajo del resto. Es un trabajo elaborado y minucioso, que requiere de mucha precisión y sincronización por parte de todos los conectores.

Como se ha comentado en el punto anterior, la empresa va a diseñar e implementar un nueva versión del sistema Plug'n Trade, desarrollando un nuevo sistema de comunicación entre mensajes más rápido, con mejor rendimiento y consiguiendo mejorar todos los tiempos de todos los conectores en lo que a latencia se refiere.

Dada la gran cantidad de módulos y conectores existentes, el módulo en el que se va centrar este proyecto, tratando de mejorarlo, será el sistema encargado de la gestión de log.

El nuevo sistema de gestión de log, al igual que todos los módulos, debe ser capaz de gestionar cada llamada de log que haga un mensaje en el menor tiempo posible, teniendo un coste operacional muy bajo y tratando de interferir con la menor cantidad de tiempo en el envío de ese mensaje. Este nuevo sistema ha de ofrecer un mejor rendimiento que el anteriormente implementado.

Uno de los principales problemas a la hora de diseñar un sistema de gestión de log es cómo reducir el coste de una operación de log cuando necesita acceder o utilizar cualquier dispositivo de entrada y salida. Estos dispositivos son los más lentos de un sistema. Su uso puede empeorar sensiblemente el rendimiento de todo el módulo y por consiguiente, de toda la plataforma en la que se integre.

Los casos más importantes a considerar son la gestión de escritura de datos en fichero y el envío de datos por el bus. Ambas operaciones son lentas y conllevan mucha latencia. Además, han de estudiarse todas las situaciones que se puedan dar en los dos casos, especialmente aquellas que impliquen mayor riesgo para el sistema, como por ejemplo situaciones de saturación o colapso de mensajes o desfallecimientos en el envío de los mensajes.

A lo largo de esta memoria, se irán desarrollando todas las funcionalidades deseadas y se explicará cómo se han ido solucionando todos los problemas encontrados hasta obtener el rendimiento óptimo y deseado.

1.3 Objetivos detallados

El principal objetivo del proyecto es el diseño, implementación y evaluación de un sistema de gestión de log para su posterior integración en un software de mensajería de baja latencia.

A continuación se enumeran los objetivos a alcanzar en este proyecto de fin de carrera:

- Evaluar y analizar en profundidad los sistemas de gestión de log actuales del mercado, estudiando todas las características de que estos disponen.
- Diseñar un sistema de gestión de log que sea capaz de mejorar el rendimiento y coste del anterior sistema implementado ya existente, ofreciendo las siguientes características:
    - **Rendimiento**. Reducción de la latencia en cada gestión de un evento de log.
    - **Capacidad**. Suficiencia para controlar cualquier situación extrema de saturación o desbordamiento de eventos de log.
    - **Flexibilidad**. Variabilidad para admitir múltiples posibilidades de configuración siendo capaz de adaptarse a cualquier situación.
- Implementar una nueva arquitectura basada en la baja latencia que sea capaz de:
    - Incrementar el número de mensajes por segundo en el sistema de gestión de log.
    - Reducir la latencia introducida a la hora de utilizar un dispositivo de entrada y salida.
    - Utilizar un serializador de datos para reducir la latencia en la trasmisión de mensajes.
- Obtener y valorar los resultados del sistema final llevado a cabo.

1.4 Definiciones

- Log: registro oficial de eventos durante un rango de tiempo en particular usado para registrar datos o información sobre quién, qué, cuándo, dónde y por qué un evento ocurre para un dispositivo en particular o aplicación.
- Sistema de Log: mecanismo estándar que se encarga de recoger los mensajes generados por los programas, aplicaciones o demonios de un sistema y enviarlos a un destino predefinido para así poder monitorizar la actividad del sistema y diagnosticar problemas.
- Appender: lugar de destino encargado de recoger y gestionar los eventos de log.
- Logback y Log4j: bibliotecas open source desarrolladas en Java que permite a los desarrolladores de software elegir la salida y el nivel de granularidad de los mensajes a tiempo de ejecución y no a tiempo de compilación como es comúnmente realizado. La configuración de salida y granularidad de los mensajes es realizada a tiempo de ejecución mediante el uso de archivos de configuración externos. [3] [4]
- SLF4J: patrón de diseño de registro de log para Java. Abstrae los diferentes frameworks, por ejemplo, java.util.logging, log4j y Logback, permitiendo al usuario final conectar el framework deseado en el momento de la implementación. [5]
- Ultra Messaging (UM): software de mensajería de última generación de baja latencia. Ultra Messaging ofrece mensajería de latencia extremadamente baja en cualquier red, además de una velocidad de transmisión muy elevada, con una fiabilidad ininterrumpida. Mejora significativamente la flexibilidad y el rendimiento de los sistemas de mensajería, reduce los costes de la infraestructura de IT y desarrolla aplicaciones comerciales que aumenten la ventaja competitiva. [6]
- Low Latency Framework Bus (LLFB): framework implementado en lenguaje Java que integra el software UM dando al usuario más flexibilidad y facilidad para su utilización y añade nuevas características y posibilidades de uso.
- Serialización de datos: proceso en el que se toman objetos y se convierte su información de estado en un formato que permita su transporte o su almacenamiento. La acción inversa consiste en deshacer la serialización y dejar el objeto con la misma forma que tenía inicialmente llamado des-serialización.

Capítulo 2. Análisis de situación

Como ya se ha mencionado, el objetivo principal del proyecto es integrar un sistema de gestión de log en un software de mensajería de baja latencia. Actualmente existen diversos sistemas de gestión de log, localizados en múltiples programas software o incluso en Sistemas Operativos. Todos ellos están programados e implementados de manera diferente y utilizan distintas técnicas para alcanzar sus objetivos.

Es necesario, antes de un profundo análisis de los principales sistemas, conocer qué es un sistema de log y cómo es su estructura. A partir de estos conocimientos, se establecerá cómo será el sistema propio que se implementará.

A continuación se definirá y estructurará brevemente qué y cómo es un sistema de gestión de log, para posteriormente realizar un análisis y estudio de las posibilidades de sistemas de log que ofrece el mercado, reflexionando sobre sus características. Finalmente, se evaluará la posibilidad de encajar algunas de sus características en nuestro propio sistema y se mejorarán o implementarán aquellos aspectos que no se ajusten a los requisitos de este proyecto.

2.1 Sistema de Gestión de Log.

El sistema de log es un mecanismo estándar que se encarga de recoger los mensajes generados por los programas, aplicaciones o demonios de un sistema y enviarlos a un destino predefinido para así poder monitorizar la actividad del sistema y diagnosticar problemas.

El uso de sentencias de log dentro del contexto de desarrollo consiste en insertar dentro de la aplicación sentencias que proporcionan información de salida útil, como puede ser en cada mensaje la fuente (el programa que generó el mensaje), la prioridad (nivel de importancia del mensaje), la fecha y la hora o múltiples datos.

Los sistemas de log para aplicaciones tienen tres componentes principales:

- Loggers [7]
- Appenders
- Layouts

Estos tres tipos de componentes trabajan juntos para permitir a los desarrolladores hacer log de los mensajes de acuerdo con el tipo y nivel de mensaje, controlar en tiempo de ejecución en el que se producen los mensajes y asignar el lugar y formato de salida.

2.1.1 Loggers.

Los Loggers son los componentes más esenciales del proceso de log, siendo los elementos responsables de capturar la información.

Los Loggers aquí considerados, son entidades con nombre. Los nombres de los loggers son sensibles a mayúsculas y siguen una regla jerárquica de nombres: un logger es un ancestro de otro logger si su nombre, seguido de un punto, es un prefijo del nombre del logger descendiente. Un logger es padre de un logger hijo si no hay ancestros entre él mismo y el logger descendiente.

Existe un logger especial llamado *rootLogger*, el cual reside en la cima de la jerarquía de loggers. Este logger tiene dos características:

- Siempre existe.
- No puede ser recuperado por nombre.

Los loggers pueden tener niveles asignados. Los niveles normales que puede tener un logger son, de menor a mayor prioridad:

- **TRACE**: Se usa para información más detallada que el nivel debug.
- **DEBUG**: Se utiliza para mensajes de información detallada que son útiles para depurar una aplicación.
- **INFO**: Se utiliza para mensajes de información que resaltan el progreso de la aplicación de una forma general.
- **WARN**: Se utiliza para situaciones que podrían ser potencialmente dañinas.
- **ERROR**: Se usa para eventos de error que podrían permitir que la aplicación continúe ejecutándose.
- **FATAL**: Se usa para errores muy graves, que podrían hacer que la aplicación dejara de funcionar.

Además hay otros dos niveles especiales que son:

- **ALL**: Tiene el nivel más bajo posible y se usa para activar todo el logging.
- **OFF**: Tiene el nivel más alto posible y se usa para evitar cualquier mensaje de log

Si a un logger no se le asigna un nivel de forma explícita, éste hereda el de su ancestro más cercano. Si no tiene ancestros, hereda el del rootLogger, que por defecto es DEBUG.

Una petición de logging está habilitada si su nivel es mayor que o igual que el nivel de su logger. De otra forma se dice que la petición está deshabilitada. El comportamiento de los loggers es jerárquico, tal y como se muestra en la Figura 2.

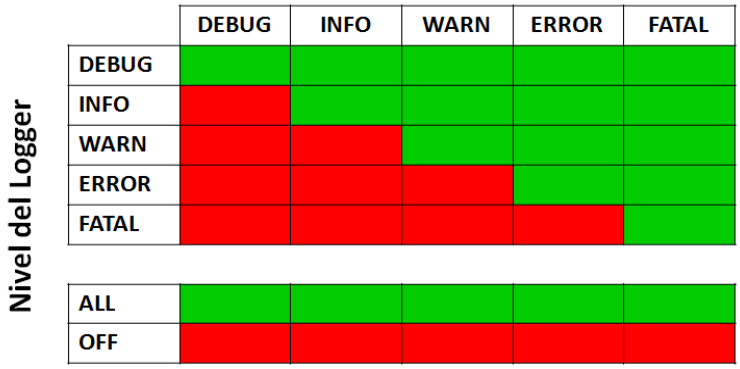


Figura 1: Nivel jerárquico de comportamiento de los loggers.

2.1.2 Appenders.

La parte que se encarga de manejar y decidir el destino de salida de cada evento de log se llama appender del sistema. Existen appenders para la consola, archivos, sockets, JMS, correo electrónico, bases de datos, etc. Un logger puede tener asociado más de un appender.

Es la parte más importarte ya que es la parte donde recae el mayor consumo de latencia en las operaciones. Es totalmente modificable y es la que más se podrá mejorar.

2.1.3 Layouts.

Son los responsables de dar formatos a los mensajes de salida, de acuerdo a lo que el usuario que esté manejado la aplicación quiera. Permiten presentar el mensaje con formato para mostrarlo en la consola o guardarlo en un archivo de texto, en una tabla HTML, un archivo XML, etc.

Los layouts también suponen una carga importante en términos de latencia ya que obtener cierta información como por ejemplo tiempos del sistema, fechas o procesos que fueron llamados para ejecutar la llamada de log supone una gran carga computacional. Este aspecto se vigilará para tratar de reducir la latencia del sistema de log final.

2.2 Análisis de Requisitos y Funcionalidades

Los principales requisitos y funcionalidades que debe ofrecer el sistema para conseguir alcanzar los objetivos son:

- El sistema debe ser capaz de gestionar una alta tasa de eventos de log por segundo. Este sistema se integrará después en un software de mensajería de baja latencia que enviará 1.000.000 de mensajes por segundo, teniendo que soportar al menos esa tasa de mensajes.

  Además, se ha de tener en cuenta que cada uno de esos mensajes puede generar uno o más eventos de log.

- La latencia total en realizar una operación completa para capturar los eventos de log no debe ser superior a los 5.000nsg incluyendo todos los costes operacionales, como pueden ser enviar eventos por el bus o escribirlos en un fichero.

  Se ha calculado que con esa latencia, el sistema de log no interferiría en el software de mensajería de baja latencia. Este y el anterior requisito, serán los dos grandes retos de este proyecto.

- Las operaciones que realice el sistema de log deben evitar ser bloqueantes. Dado que se espera que haya una gran cantidad de eventos de log, una operación bloqueante podría colapsar el sistema e incluso podría suponer la pérdida de algún evento de log.

- El sistema debe ser capaz de enviar y recibir sólo los eventos de log de nivel más importantes y descartar el resto de eventos que reciba. De esta manera se evitará el colapso del sistema en los casos peligrosos y extremos en los que exista una masiva petición de eventos de log y pueda producirse la saturación del sistema (situación que se debe evitar en cualquier caso).

2.3 Alternativas Actuales

Antes de implementar el sistema de log desde la base, se analizará en profundidad todas las variantes que existen en la actualidad de los sistemas de log, intentando descubrir si alguna de ellas ofrece alguna característica similar a la descrita en los requisitos iniciales.

Dada la complejidad existente en desarrollar completamente un sistema de log, una posibilidad que se estudiará será desarrollar un sistema de log a partir de la arquitectura de uno existente, mejorando y adaptando aquellas partes al sistema requerido.

Inicialmente se estudiará cómo están construidos, las características, ventajas y desventajas que ofrecen. La parte más interesante será analizar el rendimiento para observar si se pude adaptar al sistema de log de este proyecto. Una vez estudiadas todas las posibilidades existentes y obtenidos los resultados de rendimiento, se decidirá si son lo suficientemente óptimos como para adaptarse a este proyecto o se descartarán y se creará un sistema de log totalmente nuevo partiendo desde el inicio.

De todos los diferentes sistemas de log existentes en la actualidad, se ha decidido estudiar aquellos que están desarrollados en lenguaje Java. A pesar de que este lenguaje no es el más eficiente en cuanto a términos de rendimiento se refiere, se tomó esta decisión debido a que el proyecto global en el que irá implementado el sistema de log está completamente desarrollado en lenguaje Java y por razones de compatibilidad, se eligió esta opción.

Otras razones que se tuvieron en cuenta para elegir este lenguaje fueron:

- La flexibilidad y comodidad que el lenguaje Java ofrece al programador.
- La gestión de la memoria y la utilización que se realiza con el recolector de basura.
- La posibilidad que ofrece para generar código granulable y modulizable.
- El lenguaje Java es independiente de la arquitectura hardware, evitando que cualquier dispositivo Hardware afecte al rendimiento del sistema de log.

Por lo tanto, de los sistemas de log actuales desarrollados en lenguaje Java, se estudiarán los dos que más destacan por su calidad y usabilidad que son: Log4j y Logback.

2.3.1 Log4j

Log4j es una biblioteca open source desarrollada en Java por *Apache Software Foundation* [8] que permite a los desarrolladores de software elegir la salida y el nivel de granularidad de los mensajes o log en tiempo de ejecución y no en tiempo de compilación como es comúnmente realizado. La configuración de salida y granularidad de los mensajes es realizada a tiempo de ejecución mediante el uso de archivos de configuración externos. Log4J ha sido portada a otros lenguajes como: C, C++, C#, Perl, Python, Ruby y Eiffel.

Los appenders disponibles más destacables son:

- **AsynAppender**: Permite enviar los mensajes de log de forma asíncrona. Este appender recolectará los mensajes enviados a los appenders que estén asociados a él y los enviará de forma asíncrona.
- **ConsoleAppender**: Envía los eventos de log a System.out o a System.err.
- **FileAppender**: Envía los eventos de log a un archivo.
- **RollingFileAppender**: Extiende de FileAppender para respaldar los archivos de log cuando alcanza un tamaño determinado

Para utilizar cualquiera de sus appenders se debe configurar mediante el fichero de configuración. Al instanciar un logger, Log4J busca en la raíz del *classpath* de la aplicación el fichero llamado log4j.properties para configurarse.

2.3.2 Logback

Logback es el sucesor del proyecto log4j. Logback es más rápido y tiene una huella más pequeña que todos los sistemas de log existentes. Ofrece características únicas y útiles para un sistema de log.

Logback es una implementación concreta de SFL4J, pero podemos utilizar SFL4J con diversas librerías de logging como log4j o java.util.logging.

2.3.3 Logback vs Log4fj

Logback aporta un gran número de mejoras sobre log4j, grandes y pequeñas. Esto se debe a que Logback es conceptualmente muy similar a log4j ya que ambos proyectos fueron fundados por el mismo desarrollador. Los propios desarrolladores han realizado un profundo estudio que compara todas las características de los dos sistemas de log y determinó que se debía utilizar Logback antes que log4j [9] [10]. Las conclusiones más importantes obtenidas fueron:

- **Logback tiene implementación más rápida.**
  Los eventos de log internos de Logback se llevan a cabo diez veces más rápido en ciertas rutas de ejecución. Además tienen un menor consumo de memoria.
- **Logback-clásico utiliza la librería nativa SLF4J.**
  La clase Logger en Logback-clásico implementa la API nativa SLF4J, que no agrega latencia adicional cuando se invoca un registrador SLF4J con Logback-clásico.
- **Los archivos de configuración en XML o Groovy.**
  La forma de configurar Logback es a través de un archivo XML. Sin embargo, los archivos de configuración escritos en Groovy también se pueden utilizar [11] [12]. En comparación con XML, Groovy tiene un estilo de configuración más intuitiva, consistente y tiene una sintaxis más corta.
- **Recuperación de E / S de fallos.**
  Logback y todos sus sub-clases, pueden recuperarse de los fallos de E / S. Así, si un servidor de archivos falla temporalmente, no se necesita reiniciar la aplicación. Tan pronto como el servidor de archivos vuelve a funcionar, el appender de Logback es el encargado de recuperarse rápidamente de la condición de error anterior.
- **Procesamiento condicional de los archivos de configuración.**
  Logback soporta el procesamiento condicional en los archivos de configuración con la ayuda de elementos *if, then y Else*.
- **Filtros.**
  Logback dispone de una amplia gama de capacidades de filtrado muy superior a la que log4j ofrece.

En resumen, se han enumerado una serie de razones para elegir Logback sobre log4j y será la arquitectura que se estudiará y analizará para su posterior desarrollo y mejora.

2.3.4 Appenders de Logback

Una vez vistas las ventajas que nos ofrece Logback sobre log4, se ha decidido utilizar la arquitectura de Logback como base para desarrollar todo el sistema gestor de log.

Para desarrollar el sistema de gestión de log, debemos analizar y probablemente reconstruir la parte más importante de un sistema de estas características: el appender. Esta parte es el que decide qué hacer y dónde enviar un evento cuando es capturado. Sobre el appender recae gran parte de la latencia del sistema de log.

Logback tiene implementados varios appenders [13]. De todos ellos, se estudiarán tres appenders que se han considerado útiles (por sus características o funcionalidades) para ser posteriormente añadidos al sistema de baja latencia. Sus principales características son:

- **ConsoleAppender**
  ConsoleAppender muestra los eventos de log en la consola System.out o en System.err. Se puede especificar el formato de salida de los eventos con las diferentes configuraciones de los *encoder* (codificadores) de que dispone la arquitectura de Logback aunque deben ser especificadas por el usuario.
- **FileReaderAppender**
  Este appender escribe en fichero de manera síncrona cada vez que se produce un evento de log. Utiliza librerías de escritura estándar. Por lo tanto, el conjunto de sus operaciones pueden ser bloqueantes.
- **AsynAppender**
  Este appender utiliza una cola no bloqueante para escribir los eventos de log a fichero. El único problema es que para hacer que la cola sea no bloqueante, cuando detecta que la cola está llena al 85% de capacidad, descarta cualquier evento que reciba y no lo escribe en el fichero hasta que la cola recupere los valores estables.

2.4 Test de Rendimiento para el LLFB y los Appenders de Logback

Si se va a utilizar Logback como tecnología base del sistema gestor de log, se estudiará en profundidad el rendimiento para todas las posibilidades de funcionamiento que ofrece y para cualquier situación a la que se le someta.

Puesto que el sistema Logback ofrece una arquitectura flexible, nuestro trabajo consistiría en conocer si es posible adaptar el sistema Logback existente a los requisitos y funcionalidades especificados para nuestro sistema. Para conseguir esta tarea, se realizará una serie de test de simulación utilizando una primera versión del framework de mensajería de baja latencia que utilizará internamente el sistema Logback como motor y gestor de los eventos de log.

Al finalizar todos los test, se revisarán los resultados y se decidirá si se puede utilizar Logback como base del sistema de log que se implementará.

2.4.1 Test de rendimiento del LLFB

La primera prueba que se realizará será comprobar el rendimiento del sistema cuando envía un determinado número de mensajes por el bus de baja latencia sin capturar ningún evento de log utilizando los dos modos de envío que dispone el bus. Esta prueba tiene el objetivo de medir el rendimiento real del LLFB.

El test consistirá en que el LLFB se encargará de enviar un número de mensajes (10-100-1000-10000k) y se calculará el tiempo en mandar un número de mensajes, el número de mensajes que es capaz de enviar por segundo, la latencia media de cada mensaje y el Throughput. Antes de iniciar a tomar las medidas en el test, se

enviarán 100.000 mensajes iniciales para eliminar los tiempos introducidos por el LLFB en el momento de crear las distintas conexiones entre el emisor y el receptor. Todos los mensajes tienen un tamaño fijo de 300bytes.

El LLFB será configurado con los valores por defecto, lo que significa que la capa de transporte por la que viajarán los mensajes será TCP y lo único que variara será el modo de envío de los mensajes. El LLFB dispone de dos modos de envío:

- **Flush.** Los mensajes no son enviados hasta que el tamaño de la trama por la que viaja el mensaje está completa.
- **No-Flush**. Los mensajes son enviados sin necesidad de que la trama por la que viaja el mensaje este completa.

Los resultados obtenidos con el modo de envío Flush se muestran en la Tabla 1.

Tabla 1: Rendimiento del LLFB enviando mensajes en modo Flush.

Nº mensajes	10k		100k		1000k		10000k	
	Valor Mínimo	Valor Máximo	Valor Mínimo	Valor Máximo	Valor Mínimo	Valor Máximo	Valor Mínimo	Valor Máximo
Tiempo Total (sg)	0,1102	0,1632	1,1322	1,7221	10,859	18,0514	95,3847	168,3227
Ms/sg	87.722,17	60.114,23	87.790,23	57.820,86	92.086,13	55.396,93	104.838,5	53.100,34
Latencia media (nsg)	11.464,05	14.308,43	11.340,78	17.294,79	10.859,39	18.051,53	9.538,47	18.832,27
Throughput (Mg/sg)	27,329	19,395	24,363	16,046	25,552	15,373	29,094	14,736

Los resultados obtenidos con el modo de envío No-Flush se muestran en la Tabla 2.

Tabla 2. Rendimiento del LLFB enviando mensajes en modo No-Flush.

Nº mensajes	10k	100k	1000k	10000k	50000k
Tiempo Total (sg)	0.05709	0.607	7.0763	71,673	369,006
Msg/sg	175.155,357	164.744	141.316,021	139.520,803	135.499,056
Latencia media (nsg)	5.709,217	6.070,017	7.076,338	7.167,38	7.380,125
Throughput (Mg/sg)	48,608	45,719	39,217	38,719	37,603

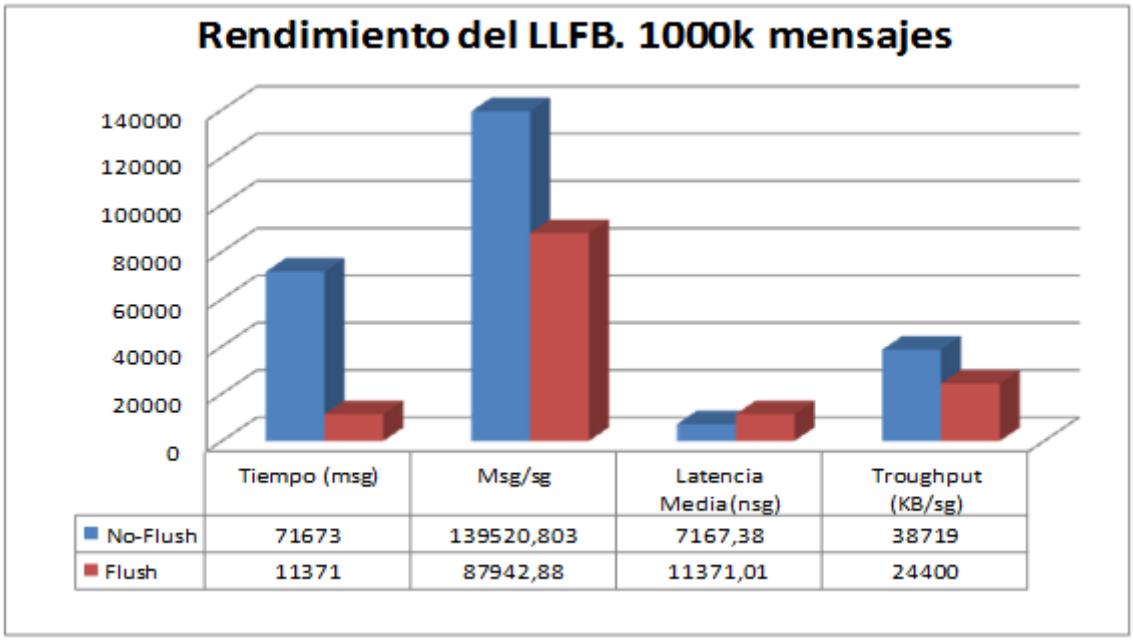


Figura 3. Resumen del rendimiento del LLFB para 1000 mensajes con los dos métodos de envío considerados.

2.4.2 Test de rendimiento del sistema UM utilizando Appenders de Logback

ConsoleAppender

Se comprobará el rendimiento del ConsoleAppender utilizándolo en el LLFB. Se quiere capturar cuál es el rendimiento que supone para un sistema estar utilizando este tipo de appender y ver si es costoso en latencia.

Es probable que este appender sea el que más tiempo requiera ya que cada evento de log supone mostrar por pantalla lo ocurrido y esta es una operación bastante costosa en rendimiento debido a que hay que realizar llamadas al sistema. Es probable que no se pueda utilizar este tipo de appender en un sistema de baja latencia ya que el coste de mostrar por pantalla los eventos de log es demasiado elevado.

Si se comprueba que los resultados son demasiado elevados, se descartará este appender y no se utilizará. En este apartado, se realizará las mismas pruebas que en el anterior apartado con el añadido de que cuando se envíe un mensaje por el bus de baja latencia, cada mensaje generará un evento de log. De este modo se captará el coste de enviar un mensaje por el bus más la latencia que se añada al hacer un evento de log, siendo el ConsoleAppender quien gestione dicho evento.

Al igual que en el punto anterior, se enviarán los mensajes por la capa de transporte TCP y se utilizarán los dos modos de envío: Flush y No-Flush.

Los resultados obtenidos con el modo de envío Flush se muestran en la Tabla 3.

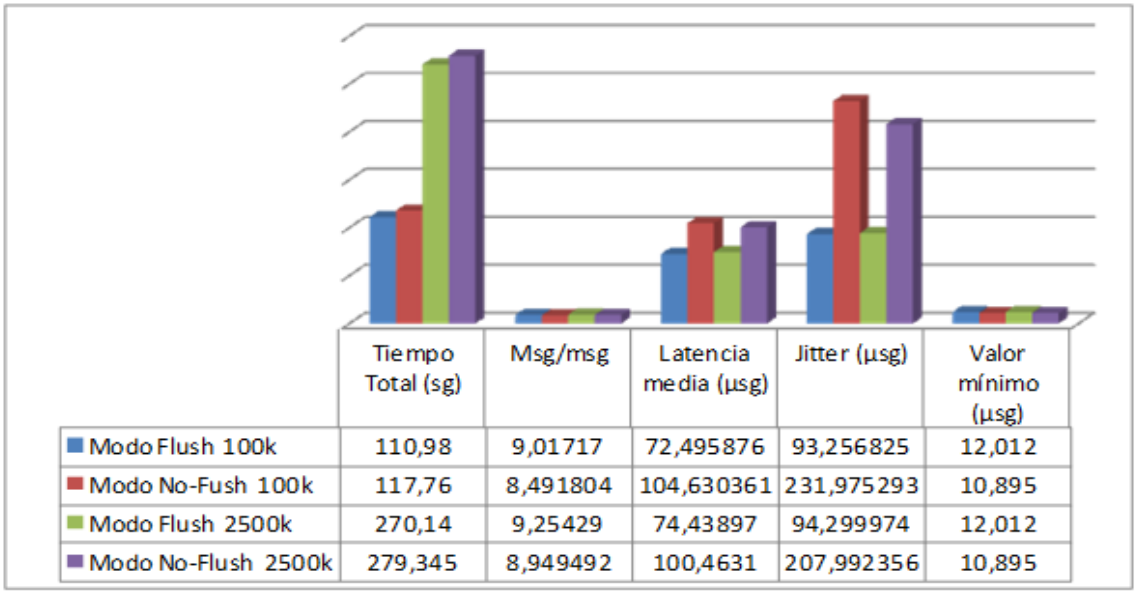
Tabla 3. Rendimiento del LLFB utilizando el appender ConsoleAppender enviado los mensajes con el modo Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	1,14	11,38	110,98	270,14
Msg/sg	8.698,36	8.779,33	9.017,17	9.254,29
Latencia media (µsg)	71.634	71,355	72,495	70,438
Jitter (µsg)	80,952	76,786	93,256	94,299
Mínimo (µsg)	12,571	12,012	12,012	12,012

Los resultados obtenidos el modo de envío No-Flush se muestran en la Tabla 4.

Tabla 4. Rendimiento del LLFB utilizando el appender ConsoleAppender enviado los mensajes con el modo No-Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	1,156	11,455	117,760	279,345
Msg/sg	8.648,54	8.729,137	8.491,804	8.949,492
Latencia media (µsg)	102,153	101,267	104,630	100,466
Jitter (µsg)	194,606	123,909	231,975	207.992,356
Mínimo (µsg)	15,085	14,806	10,895	10,895



	Tiempo Total (sg)	Msg/msg	Latencia media (µsg)	Jitter (µsg)	Valor mínimo (µsg)
Modo Flush 100k	110,98	9,01717	72,495876	93,256825	12,012
Modo No-Fush 100k	117,76	8,491804	104,630361	231,975293	10,895
Modo Flush 2500k	270,14	9,25429	74,43897	94,299974	12,012
Modo No-Flush 2500k	279,345	8,949492	100,4631	207,992356	10,895

Figura 4. Resumen del rendimiento del appender ConsoleAppender utilizando el LLFB para distinta cantidad de mensajes con los dos métodos de envío considerados.

FileReaderAppender

Se comprobará el rendimiento del appender FileReaderAppender utilizando el LLFB. Se quiere capturar cual es el rendimiento que supone para un sistema estar utilizando este tipo de appender y ver si es costoso en latencia.

Este appender escribe en fichero todos los eventos de log que se registren en la aplicación. Lo hace de manera síncrona, es decir, por cada evento de log que se genere, el siguiente evento debe esperar a escribir en el fichero hasta que el primero termine. Esta es una operación muy costosa ya que bloquea la aplicación que se esté utilizando hasta que la operación de escritura en fichero.

Se comprobará el verdadero rendimiento que ofrece utilizar este appender con el bus de baja latencia y se comprobará cuanto tiempo supone el bloqueo. Al igual que en punto anterior, se enviarán los mensajes por la capa de transporte TCP y se utilizarán los dos modos de envío Flush y No-Flush.

Los resultados obtenidos con el modo de envío Flush se muestran en la Tabla 5

Tabla5. Rendimiento del LLFB utilizando el appender FileReaderAppender enviado los mensajes con el modo Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	0,54	5,337	46,919	525,42
Msg/sg	1.8343,42	1.8597,512	2.1313,11	1.9032,26
Latencia media (µsg)	32,150	31,345	25,487	30,831
Jitter (µsg)	2.729,676	3.393,169	6.100,216	2.755,5502
Mínimo (µsg)	17,042	16,203	15,365	15,365

Los resultados obtenidos con el modo de envío No-Flush se muestran en la Tabla 6.

Tabla6. Rendimiento del LLFB utilizando el appender FileReaderAppender enviado los mensajes con el modo No-Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	0.243	2,498	24,916	251,075
Msg/sg	41.146,323	40.025,908	40.133,722	39.993,6644
Latencia media (µsg)	18,456	19,088	18,633	18,816
Jitter (µsg)	1.670,324	2.075,683	3.724,216	4.2734,761
Mínimo (µsg)	15,085	14,806	14,806	14,806

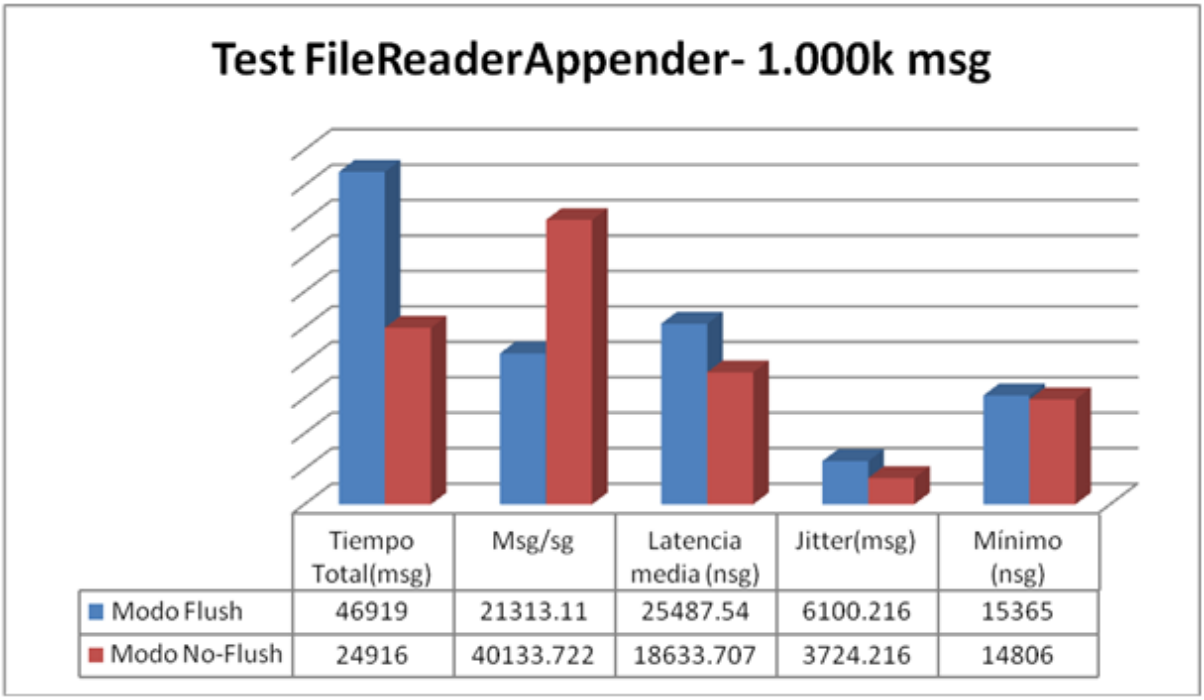


Figura 5. Resumen del rendimiento del appender FileReaderAppender utilizando el LLFB para 1000 mensajes con los dos métodos de envío considerados.

AsynAppender

AsynAppender hace el registro de log de forma asíncrona. Actúa sólo como un distribuidor de eventos y por lo tanto debe hace referencia a otro appender para hacer algo útil. Con el fin de evitar la pérdida de eventos de registro, este appender debe estar cerrado. Es responsabilidad del usuario cerrar appenders, típicamente al final del ciclo de vida de la aplicación.

Por defecto los eventos se almacenan en una *BlockingQueue* modificada. Un subproceso de trabajo creado por AsynAppender toma eventos de la cabeza de la cola y los envía al appender sencillo unido a AsynAppender. Por defecto, AsynAppender perderá los eventos de nivel TRACE, DEBUG INFO si la cola está de 80% de su capacidad. Esta estrategia tiene un efecto sorprendentemente favorable sobre el rendimiento a costa de pérdida de eventos.

Al igual que en punto anterior, se enviarán los mensajes por la capa TCP y se utilizarán los dos modos de envío Flush y No-Flush. Los resultados obtenidos con el modo de envío Flush se muestran en la Tabla 7.

Tabla7. Rendimiento del LLFB utilizando el appender AsynAppender enviado los mensajes con el modo Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	0,33	2,84	30,106	74,086
Msg/sg	29.630,057	35.188,02	33.215,33	33.744,47
Latencia media (µsg)	8,751	8,105	8,266	8,180
Jitter (µsg)	4.679,086	6.020,877	7.480,839	9.142,782
Mínimo (µsg)	4,190	3,911	3,631	3,631

Los resultados obtenidos con el modo de envío Flush se muestran en la Tabla 8.

Tabla8. Rendimiento del LLFB utilizando el appender AsynAppender enviado los mensajes con el modo No-Flush.

Nº mensajes	10k	100k	1000k	10000k
Tiempo total (sg)	0,1635	1,599	15,652	158,120
Msg/sg	61.136,654	62.511,853	63.885,740	63.242,889
Latencia media (µsg)	7,651	6,709	6,043	6,123
Jitter (µsg)	10.530,389	8.917,893	1.1969,398	1.4007,087
Mínimo (µsg)	3,632	3,073	2,514	2,514

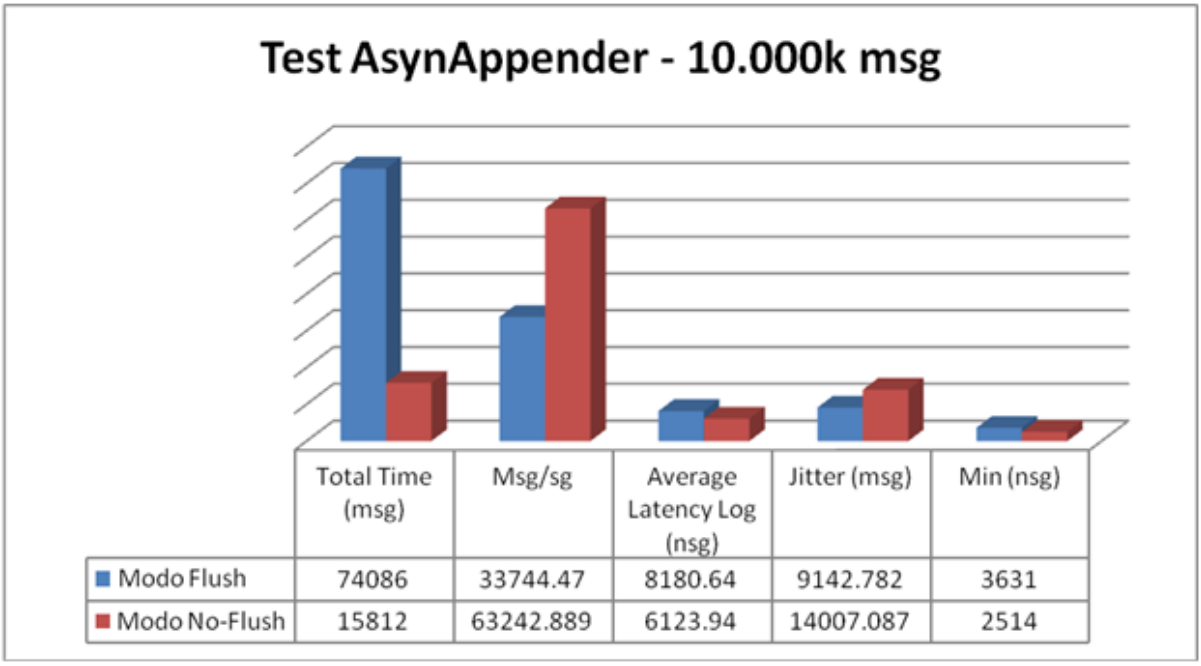


Figura 6. Resumen del rendimiento del appender AsynAppender utilizando el LLFB para 10.000 mensajes con los dos métodos de envío considerados.

Finalmente, a modo de resumen, se deja constancia en las figuras 7 y 8 una comparación de todos los appenders estudiados.

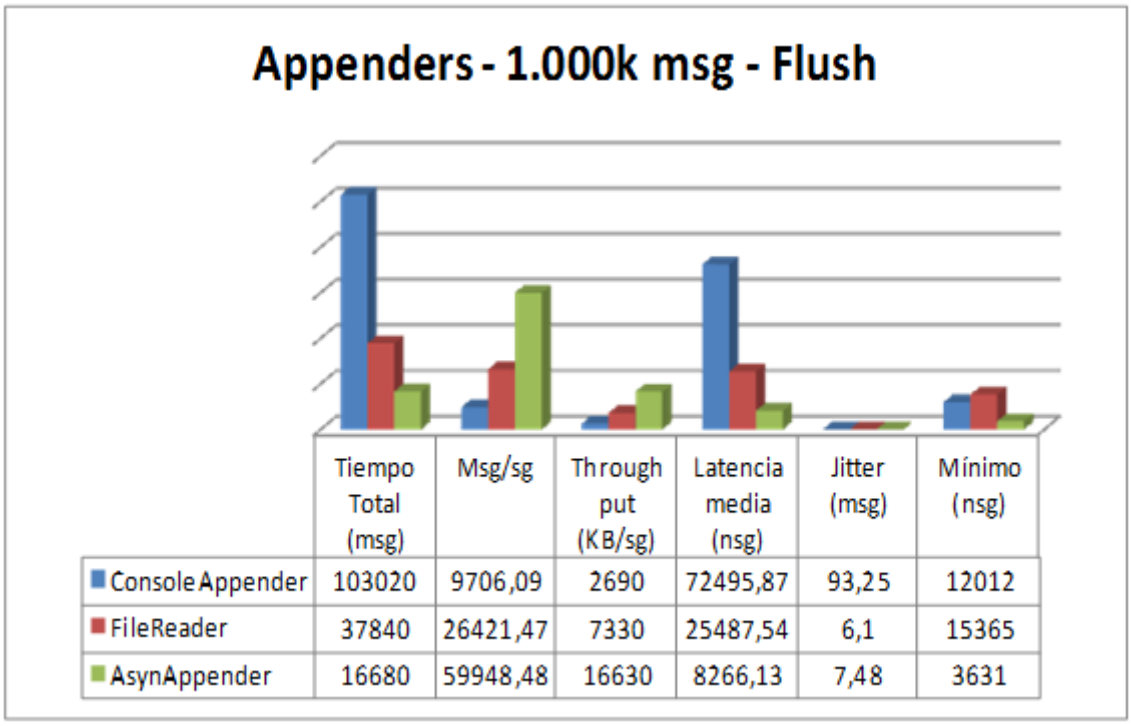
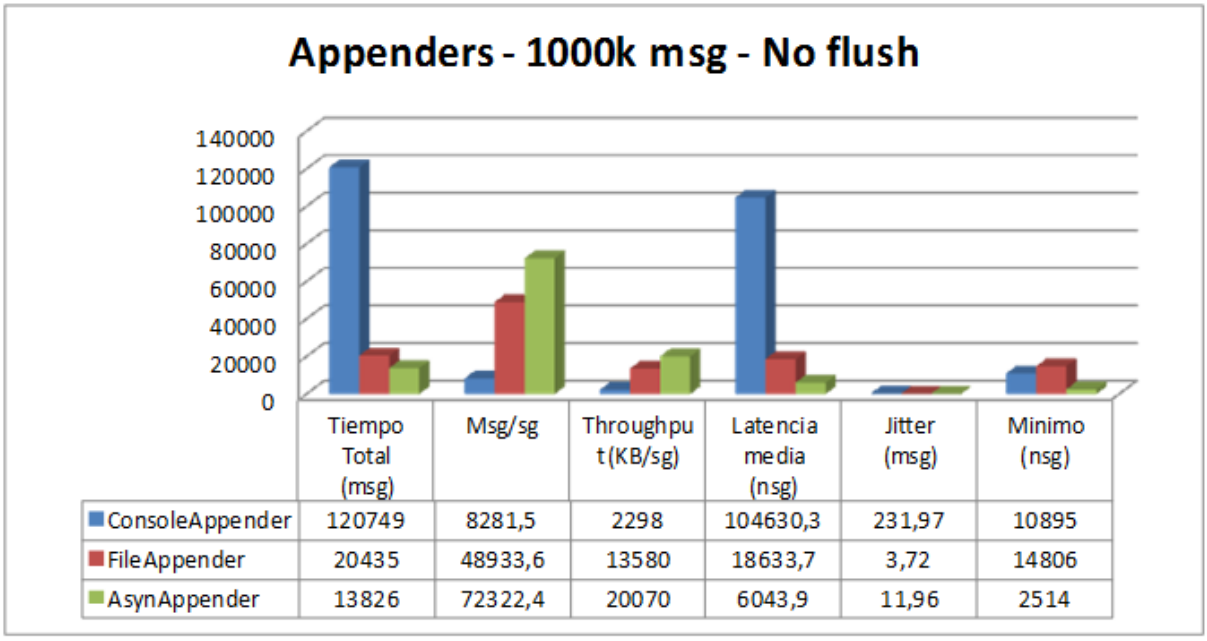


Figura 7. Resumen del rendimiento de los tres appenders utilizando en el LLFB para 1000 mensajes con el método de envío considerado anteriormente Flush.



	Tiempo Total (msg)	Msg/sg	Throughput (KB/sg)	Latencia media (nsg)	Jitter (msg)	Minimo (nsg)
ConsoleAppender	120749	8281,5	2298	104630,3	231,97	10895
FileAppender	20435	48933,6	13580	18633,7	3,72	14806
AsynAppender	13826	72322,4	20070	6043,9	11,96	2514

Figura 8. Resumen del rendimiento de los tres appenders utilizados anteriormente con el LLFB para 1000 mensajes con el método de envío considerado anteriormente No-Flush.

2.4.3 Resultados de los test

LLFB

- Se han realizado 5 repeticiones por cada test ya que los resultados varían de una prueba a otra. Los resultados siempre han estado en un rango mínimo (MIN) y máximo (MAX). Cuando no aparece en las tablas dichas diferencias, es que siempre los resultados han sido muy similares y dicha diferencia es despreciable.
- La explicación para la diferencia entre MIN – MAX se debe a que cuando se realiza la primera prueba, el sistema operativo (S.O.) no tiene guardada ninguna variable ni valor en memoria cache. Al ejecutar en una siguiente ocasión, el S.O. mantiene las direcciones de memoria en cache y los accesos son más rápidos a las variables compartidas, por lo que los resultados son mejores.
- En todas las pruebas, ha habido una fase de calentamiento (envío de 100.000 mensajes) para estabilizar lo máximo posible el LLFB.
- La latencia, el Throughput y los mensajes por segundo se estabilizan a partir del envío de aproximadamente 1000k. Inicialmente se obtienen valores más altos (valores buenos), pero a medida que se envían mensajes se va estabilizando en un valor y dicho valor se va manteniendo pudiendo oscilar un 5%.
- El modo de envío de mensajes No-Flush es más eficiente. Mejora en todos los aspectos (tiempo, mensajes por segundo, Latencia, Throughput) al modo de envío Flush entre un 30-35%, pero en contra partida, utilizar este modo de envío puede perder algún mensaje, mientras que el modo Flush es más seguro.

ConsoleAppender

- Los resultados nos muestran que el ConsoleAppender en modo No-Flush es menos eficiente en cualquier término de latencia o rendimiento. Esto se debe a la gran cantidad de mensajes que manda y que la consola no es capaz de gestionar y se ve desbordada.
- Escribir por consola es más costoso que escribir en fichero y tiene una latencia media muy superior a la exigida como requisito para un sistema de baja latencia (más de 70.000nsg frente a los 5.000nsg).
- Se descartará la utilización en el sistema de baja latencia que se implementará debido a la alta latencia que este appender genera.

FileReaderAppender

- El uso de FileReaderAppender en modo No-Flush es más eficiente que en modo Flush en cualquiera de los aspectos.
- Los Jitter siempre pueden variar ya que depende mucho de las operaciones que haga el sistema operativo en ese momento.
- De todas las pruebas, se puede observar cual es el tiempo mínimo en hacer una escritura de log fichero de manera síncrona que en este caso es de 14.806nsg.
- El tiempo conseguido para escribir en fichero es demasiado alto y sobrepasa las expectativas para utilizarlo en el sistema de baja latencia (más de 15.000nsg frente a los 5000nsg de latencia media máxima permitida).
- Se tendrá que implementar un nuevo appender que haga esta misma operación síncrona de manera más eficiente ya que es necesario tener una operación de escritura en fichero de forma segura y sin descarte de eventos de log en el sistema de gestión de log de baja latencia.

AsynAppender

- Los resultados nos muestran que utilizar el AsynAppender con una cola de 1000 descarta demasiados log, de ahí la razón de que los ficheros ocupen menos espacio y consiga mejores latencias.
- El uso del AsynAppender es más eficiente y de menor latencia en todos los aspectos de rendimiento comparado con FileReaderAppender.
- Necesitamos una cola de más de 100.000 elementos y utilizar la opción “No descartar” para no perder los eventos de log que llegan. Esta opción supone la utilización de demasiada memoria y aumenta la latencia general del sistema.
- El AsynAppender en modo No-Flush es más eficiente y de menor latencia que en modo Flush ya que consigue enviar más mensajes por segundo, mejorando las latencias y el rendimiento
- A pesar de que mejora notablemente los resultados del FileReaderAppender, los resultados respecto a la latencia no son lo suficientemente aceptables como para implementarlo en el sistema de baja latencia (6.000nsg frente a los 5.000nsg permitidos de latencia media máxima permitida).
- Se tendrá que implementar un nuevo appender que mejore las operaciones de este appender y además evite descartar los eventos de log recibidos ya que es necesario disponer de una operación que escriba en fichero de manera asíncrona sin bloquear al sistema.

Capítulo 3. Diseño y modelo de desarrollo

Como se ha estudiado en el capítulo anterior, los appenders que ofrece Logback no cumplen los requisitos necesarios para poder ser integrados dentro del software de mensajería de baja latencia, por lo que se ha decidido implementar y desarrollar las partes modificables de la arquitectura de estos para poder adaptarlas e integrarlas en el proyecto y lograr cumplir los objetivos propuestos.

Por lo tanto, será necesario elaborar 3 nuevos appenders a partir de los anteriormente estudiados. Una vez finalizada esta tarea, se estudiará si es necesario reimplantar la parte del Layout de Logback, que es la encargada de escoger que parámetros de información que se van a capturar en los eventos de log o por el contrario, se mantendrá la que ya existe.

El desarrollo de los nuevos appenders conllevará desarrollar e implementar nuevos mecanismos para poder reducir la latencia que estos producen actualmente. Tendremos que modificar el diseño y estructura de los dispositivos de entrada y salida, modificar aspectos del bus de baja latencia o desarrollar una nueva forma de serializar los datos para conseguir reducir la latencia a la hora de enviar los eventos por el bus y de ese modo conseguir el rendimiento deseado.

3.1 Modelo evolutivo en espiral

Todo el diseño del sistema de gestión de log e implementación del mismo se realizará siguiendo el *modelo evolutivo en espiral* [14] [15]. El software se irá desarrollando de forma cíclica. Las etapas se irán repitiendo cada cierto tiempo, con la ventaja de que en cada una de ellas se irán mejorando o cumpliendo una serie de objetivos hasta conseguir alcanzar la solución final deseada.

Breve descripción del modelo llevado a cabo.
En cada ciclo o etapa se fijan unos objetivos, se analiza el riesgo que conlleva cumplir dichos objetivos, se planifican y se desarrollarán para al final de la etapa verificar y probar los resultados obtenidos para poder pasar a la próxima etapa.

Justificación del modelo elegido.
Se ha elegido este modelo ya que existen algunos requisitos iniciales que siempre han de cumplirse. Por ejemplo, las medidas de latencia han de estar siempre por debajo del máximo permitido para un sistema de baja latencia o el cálculo de las operaciones totales no puede superar, modificar o interferir en LLFB.

Este modelo nos ofrece desarrollar el sistema de gestión de log siempre de igual manera, lo que supone adquirir experiencia y conocimientos a medida que se va avanzando en cada etapa. Las ideas con las que se empiezan en cada etapa son siempre las mismas:

- Siempre han de cumplirse los requisitos iniciales.
- Cada ciclo debe añadir una o varias nuevas funcionalidades que serán extraídas de los objetivos con los que se comenzó el proyecto para posteriormente desarrollarse e integrarse con los objetivos anteriores.
- Al final de cada etapa, se verificará que las nuevas funcionalidades no afectan a lo ya implementado y que no se deja de cumplir alguno de los objetivos del proyecto.

Además, se tuvo en cuenta que en cualquier proyecto siempre es probable que aparezcan nuevos requisitos y/o funcionalidades y que haya que cambiar alguno de ellos a medida que se va diseñando ya que no siempre se podrá ir por el camino deseado al encontrar dificultadas inesperadas. Entonces, puede que sea necesario volver hacia atrás en alguna etapa o adaptarlo en la actual para poder seguir progresando.

Por esta razón se justifica la elección de este modelo de desarrollo ya que permite volver pasos hacia atrás en el desarrollo, facilitando readaptar el proyecto a las nuevas modificaciones.

Riesgos y dificultades que el uso de este modelo conlleva.
El mayor riesgo que existe es que al añadir una nueva funcionalidad o tratar de cumplir un nuevo objetivo se aumenta la latencia ya sea bien del appender que se esté implementando o del LLFB.

Puede haber nuevos objetivos que empeoren los ya conseguidos, por lo que se prestará especialmente atención en la última fase de cada ciclo que la implementación de los nuevos objetivos no empeora los resultados de rendimiento con los que se empezó al principio de la etapa.

A todo lo anterior dicho, se debe tener en cuenta que nuestro software irá integrado en un software de baja latencia y que en muchos momentos utilizará algunas de sus funcionalidades. Esto implica un requisito más, que es la dependencia de las características de este software con el nuestro. Por lo tanto, cuando cambie algunas de las características del software del proyecto principal de las que depende este proyecto, se deberá adaptar y cambiar las cosas que se necesiten para su acoplamiento y correcto funcionamiento en el proyecto principal.

3.2 Descripción de requisitos

En el capítulo 2 se explicaron cuáles iban a ser los principales requisitos y funcionales que tenían que cumplir los appenders en Logback para estudiar si serían útiles a posteriori. Una vez analizados los resultados y comprobada su ineficiencia, se tratará de implementar unos nuevos cumpliendo con los requisitos planteados.

La idea de diseño estará enfocada en crear unos nuevos appenders que sean capaces de manejar eficientemente los mensajes que se producen en un evento de log, resultando un coste operativo muy bajo en términos de latencia o consumo de recursos.

Los appenders de baja latencia tendrán que estar preparados para manejar un gran número de mensajes de log por segundo, logrando un rendimiento óptimo ideal (alto rendimiento, baja latencia) para cualquier condición a la que sean sometidos.

Han de ser capaces de almacenar los mensajes en la memoria, escribir o enviar eventos de log a un archivo y recibirlos cuando son enviados. Para conseguir una baja latencia y un alto rendimiento se tendrán que utilizar técnicas como buffers de memoria o mecanismos de propósito general y habrá que proponer una solución de programación concurrente para la integración en el bus de baja latencia.

Todos los appenders han de ser fácilmente configurables a través de un archivo de configuración, dando al usuario una amplia gama de posibilidades para operar y administrar todos los mensajes de log que se produzcan en su aplicación.

3.3 Esquema de desarrollo

Una vez desglosadas las ideas generales de lo que se debe realizar y cumplir, se esquematizarán y dividirán las tareas para ir desarrollando organizadamente todos los componentes. Se han separado por una parte, cuáles han sido los requisitos y objetivos restrictivos para cualquier etapa del proyecto y que siempre deberán cumplirse.

Por la otra parte, se han mencionado cuáles fueron los requisitos y funcionalidades que se fueron incorporando en cada etapa de desarrollo a medida que se iban verificando todos los anteriores. Todos estos requisitos y características se irán explicando en los capítulos posteriores.

Requisitos y características iniciales

- El sistema de gestión no podrá empeorar el rendimiento del LLFB en ningún aspecto de rendimiento y se debe adaptar a cualquier cambio que surja en el LLFB. (1.000.000 millón de mensajes por segundo).
- El cómputo de todas las operaciones nunca puede superar la latencia base (5.000nsg). Por tanto, las operaciones que se realicen se han de implementar de forma óptima utilizando las técnicas programación eficiente.
- Los appenders han de mejorar en términos de rendimiento y latencia a su antecesor appender de Logback.

Desglose de requisitos y características

- Diseñar por completo 3 nuevos appenders que al menos igualen en características y superen en rendimiento a los appenders de Logback.
    - Implementar de manera eficaz la captura de los eventos de log de forma que suponga el mínimo gasto en latencia.
    - Mejorar la forma en la que se utilizan los dispositivos de entrada y salida para escribir a fichero.
    - Incorporar todas las características y opciones modificables de los appender de Logback y añadir las siguientes características personalizables para el usuario:
        - Eliminar log en caso de saturación (descarte).
        - Elegir el tamaño de los diferentes artefactos que se utilicen para gestionar la memoria eficientemente.
        - Permitir solo recoger ciertos eventos de log fijados por su nivel de prioridad.
- Desarrollar otro appender que sea capaz de utilizar el bus de baja latencia para enviar los eventos de log producidos en el LLFB y los eventos que un usuario genere en una aplicación.
    - Implementar un demonio que reciba cualquier evento de log del LLFB, gestionándolo de manera eficiente.
    - Desarrollar un método que sea capaz de serializar y des-serializar los datos que viajan por el bus de manera eficiente.
    - Aplicar las mismas características personalizables que los appenders anteriormente generados
    -

En el siguiente capítulo, se describirán las características, implementación interna y los modos de funcionamiento para cada appender. En los últimos capítulos, se incluirán las pruebas de rendimiento y una tabla comparativa como resumen de los resultados de todos ellos.

Capítulo 4. Implementación de la solución propuesta

A lo largo de este capítulo, se explicará el desarrollo de cada una de las partes del sistema de gestion de log y cómo se han ido implementando los nuevos appenders. Todo ello ha sido programado y desarrollado utilizando el lenguaje Java por razones de compatibilidad con el resto de componentes del sistema de trading. Además, al ser un lenguaje dinámico, independiente de la plataforma de desarrollo y arquitectura del computador, orientado a objetos y con soporte para multihilos, se adapta perfectamente a las necesidades de este proyecto.

Se programarán nuevos appeders con los componentes y opciones requeridos y se modificarán los Layout y de la arquitectura de Logback, quedando una estructura totalmente nueva a partir de la ya implementada en Logback.

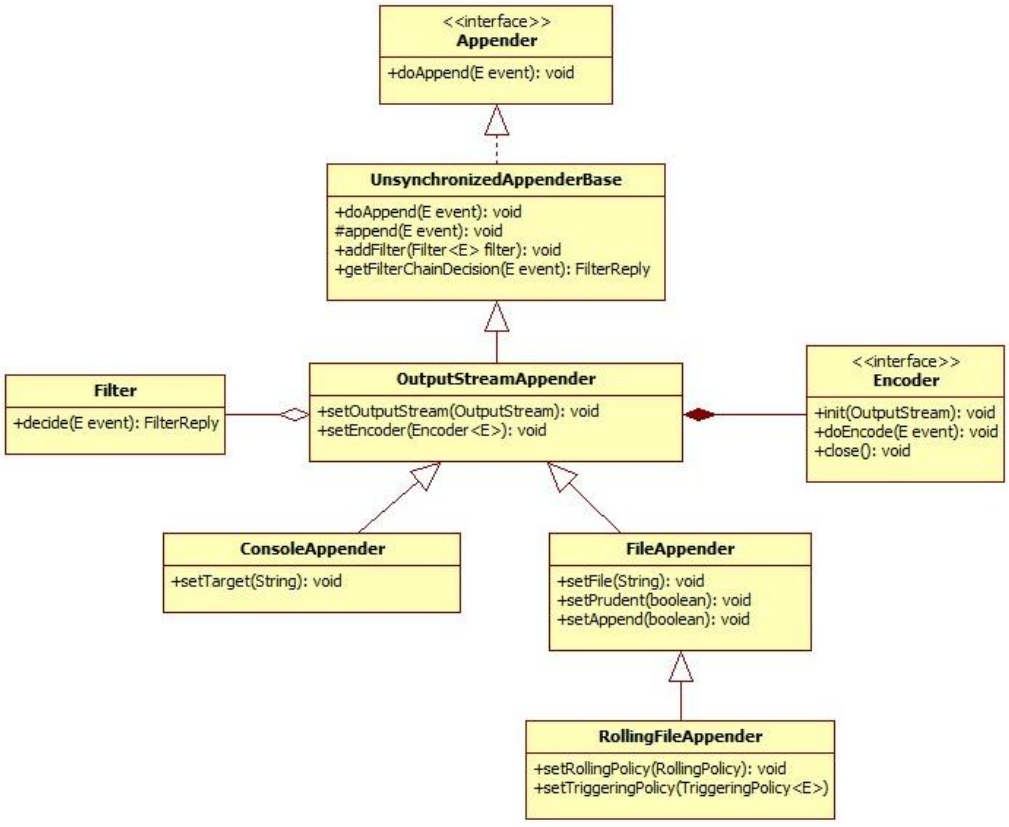


Figura 9. Diagrama de clases de de los appenders de Logback

Como muestra la figura 9, se partirá de la interfaz *Appender* para desarrollar los appenders deseados ya que esta estructura es la base sobre la que se asientan los appenders y es totalmente modificable. El resto de clases que se muestran heredan de *Appender* y son los distintos appenders que hay en logback. Cuando se crea una clase que implementa esta interfaz, se dispone del metodo *doAppend*, que es el encargado de capturar los envetos de log.

Este método es el que mas latencia generá, por lo que se diseñará un método totalmente nuevo capaz de reducir y mejorar el rendimiento. Dado que cualquiera de los appenders que se va a implementar va a tener la posibilidad de elegir multiples configuraciones, al igual que en Logback, para el correcto funcionamiento de cualquiera de ellos, deberán ser ejecutados e inicializados con un fichero de configuración. Esto facilitará la configuración, tanto en la parte del programador, como para el uso posterior del usuario.

4.1 Modelo de programación

Todas clases, métodos y atributos que se implementen deberán ser áltamete eficientes y requerirán de una programación muy exigente para obtener los términos de latencia deseados, donde habrá que buscar el máximo rendimiento a cada línea insertada en el código.

A pesar de utilizar un lenguaje de alto nivel, se tratará de optimizar el código y acercarnos más un lenguaje de programación de bajo nivel vigilando todas las sentencias que afectan a la memoria, la sincronizción de procesos activos, bloqueos, la ordenación del código, el compilador etc.

Estas características obligarán a tener en cuenta muchos aspectos en la programación y desarrollo del codigo. Para conseguir cumplirlas con precisión y eficacia, se seguirá el modelo de programación eficiente para aplicaciones de baja latencia y programación concurrente [16]**.**
Las principales directrices que se seguirán son:

- Hacer los objetos, métodos y clases **inmutables.**
- Utilizar la cláusula **final** para definir aquellas clases, métodos y variables que no vayan a cambiar su contenido durante la ejecución. Esto facilita la labor del compilador.
- Intentar no utilizar o reducir métodos **synchronized**. [17]
- Utilizar **Locks,** utilizando la estrategia del bloqueo injusto "**unfair locking strategy**" [18][19]
- Usar **profilers** para analizar cómo se desarrolla el código y optimizar las partes que más CPU consumen en el código. JProfiler es un software que analiza todos estos aspectos para aplicaciones Java [20].
- Utilizar la ley Amdhal y encontrar la ruta de ejecución secuencial [21].
- Evitar el **context-switching** (demasiados hilos consumen demasiados recursos, evitar programar con más hilos que cores dispone el procesador)
- Evitar **boxing, unboxing (castings de los tipos de valores)**
- Reducir el uso de memoria ayudará al recolector de basura "**GC"**, en particular, si se puede reducir el número de objetos de mediana edad. También evitar la generación permanente de objetos o variables.

Otras consideraciones y estilo de programación que se van a tener en cuenta para desarrollar el código son:

- Utilizar bucles que no superen un máximo de complejidad O(n) al igual que todos los métodos y librerías que se utilicen.
- Eliminar la programación recursiva.
- Limitar todo el contenido de los campos para que ocupen la menor cantidad de bytes posible.
- Seleccionar, siempre que sea posible, los tipos de datos mas eficientes para el compilador (Char en vez de String, Long o Integer en lugar de Double, o StringBuilder en lugar de String para cadenas largas).
- Emplear estructuras de datos áltamente eficiente, olvidándonos de estructuras de datos como listas enlazadas (LinkedList) o vectores (ArrayLists) [22] por el consumo que hacen de la memoria y utilizar estructuras mas eficientes como HashMap [23] o Disruptores [24][25].
- Enviar mensajes por el bus de baja latencia previamente serializados utilizando técnicas de serialización de datos.

- Crear y destruir manualmente los objetos para mejorar el trabajo del recolector de basura de Java para evitar saturar la memoria [26].
- Controlar la creación de hilos y se comprobará su correcta finalización, evitando posibles bloqueos o interbloqueos. [27] [28].

4.2 Modelo de eventos de log

El primer problema que se ha tenido que superar ha sido adaptar la arquitectura y la API de Logback a la que se va a querer implementar. Logback dispone de interfaces totalmente modificables que permiten adaptar su arquitectura base.

La parte más importarte a resolver es el momento de capturar un evento de log cuando éste se produce. Por una parte, se debe tratar de capturar la mayor cantidad de información posible sin que ello suponga un coste operacional muy alto. Y por otra parte, se debe tratar de unir toda esa información y gestionarla como si fuese un solo bloque.

El modelo de eventos de log se encargará de registrar toda la información de cada evento cada vez que se realiza una llamada desde una aplicación. Por cada evento se creará una instancia que se ha llamado "*LogEvent*" y que contendrá toda la información de ese evento.

Ya que la captura de cada parámetro de información es costosa en latencia, se va a implementar que los appenders puedan elegir la información a capturar y luego trabajar con la información seleccionada. La información máxima que se puede obtener por cada evento de log es:

- Contenido del mensaje de log – información configurable por el usuario en la aplicación.
- Nivel de log – nivel de riesgo e importancia del mensaje generado.
- Nombre del hilo que realiza la llamada de log
- Logger – nombre del logger con el que fue registrado el evento de log
- Fecha y hora exacta en milisegundos
- Modo de registro de eventos – modo del appender que se está utilizando.

Para poder escoger qué información almacenar de un evento de log, se han creado 3 modos diferentes de eventos que permitirán seleccionar la información deseada. Unos modos serán más eficientes que otros en función del número de parámetros de información que capturen. Los 3 modos disponibles son:

- **Only_Message**: recoge la información únicamente en relación con: mensaje de log y el nivel de log.
- **AllComplete**: recoge toda la información relativa a: mensajes del evento de log, nivel de log, el nombre del subproceso de llamada, nombre de la aplicación que realiza el evento, tiempo de ejecución en el que se hizo log y la fecha actual en milisegundos
- **LogbackLayout**: recoge toda la información capturada utilizando un layout de Logback. Este modo dará la posibilidad de usar los patrones de layout que implementa Logback [29]. Aunque el rendimiento decaerá, se decidió que podía ser una posibilidad interesante para dotar de mayor versatilidad a los appenders.

Después de capturar los parámetros con la información de un evento de log, el appender operará con ellos y los gestionará de maneras diferentes:

- Los eventos de log se pueden almacenar en un buffer circular.
- Escribir directamente la información de cada evento en un archivo.
- Enviar el contenido del evento por el bus de baja latencia.

Esto dependerá del appender de baja latencia y el modo de escritura (síncrono o asíncrono) que se utilice. Todas estas posibilidades serán explicadas en los siguientes apartados.

Una vez estructurado el modelo para recoger los eventos de log, se procederá a la implementación de los appenders. Logback posee por separado dos appenders, uno síncrono y otro asíncrono que escriben el contenido de los eventos de log a fichero. Al ver que utilizan diferentes operaciones para recoger los eventos pero su finalidad es la misma, se ha decidido implementar un appender que integre ambos y mejore su rendimiento a la hora de escribir en fichero.

Posteriormente, se diseñará otro appender que utilice las mismas técnicas de rendimiento pero en esta ocasión será utilizado en el framework de baja latencia. De este modo, se dispondrá de dos appenders con múltiples posibilidades, uno de ellos será capaz de escribir en fichero eficientemente y el otro será capaz de utilizar el bus para enviar los eventos y después dar la posibilidad al usuario de que los capture y gestione como desee, incluso escribir en fichero utilizando el primer appender.

4.2 Appender Híbrido: HybridFileAppender

HybridFileAppender ha sido diseñado para recoger los eventos de log que se produce dentro de una aplicación y escribir los parámetros de información del los eventos en un archivo. La principal característica con la que se dotará a éste appender será la posibilidad de escribir en fichero de dos formas diferentes: síncrona y asíncronamente.

Este appender va a implementar una versión que agrupará los dos appenders de Logback y mejorará cada uno de ellos. Además implementará una tercera versión que será mixta, que consistirá en utilizar dos modos de envió, con la idea de que se pueda utilizar uno u otro en función de las necesidades de la aplicación que se esté utilizando. Si la aplicación se encuentra en una situación liberada, se podrá usar el modo síncrono y si el sistema se encuentra en una situación de saturación, se cambiará al modo asíncrono.

Las características y requerimientos finales que se han provisto a este appender para que cumpla con los requisitos que se habían planteado inicialmente son:

- Gestionar y capturar eficientemente de los eventos de log para su posterior escritura a fichero.
- Manejar los parámetros de información sin interferir en el rendimiento del sistema, sin añadir bloqueos y dando la posibilidad de elección y selección de parámetros al usuario. Todos los parámetros han de ser configurados mediante un fichero de configuración
- Realizar todo el conjunto de operaciones de manera síncrona o asíncrona.
- Evitar saturar los tamaños de los buffers para almacenar los eventos de log, especialmente en situaciones de saturación.
- Diseñar alternativas para gestionar los eventos de log en caso de que el appender se encuentre saturado.
- Compatibilidad con las funcionalidades de Logback para que el usuario final pueda utilizarlas, siendo consciente de la perdida de rendimiento que eso supondría, pero en algunos casos concretos, podría interesar.
- Utilización de un fichero de configuración para la puesta en marcha y selección de todos los parámetros por el usuario final.

4.2.1 LogEvent

Siguiendo el modelo de eventos de log creado inicialmente, se ha creado uno que se adapte a las características mencionadas anteriormente para este appender. Se construyó la clase LogEvent cuya

funcionalidad consiste en que cuando un evento de log es capturado, este appender genera una nueva instancia *LogEvent* que captura un número determinado de parámetros en función del modo de evento de log. Cualquier instancia creada devuelve una nueva instancia de *LogEvent* con todos los parámetros inicializados.

La clase *LogEvent* puede obtener información al capturar un evento de log desde una aplicación en tres modos diferentes, que son los mismos que en el modelo general. Estos modos se diferencian entre sí por la cantidad de información contenida en cada modo y por los métodos distintos que tendrán accesibles cada uno de los modos. El modo de envío de los eventos de log se seleccionará desde el archivo de configuración.

4.2.2 Implementación

Se ha agrupado todo el conjunto de clases en un módulo llamado *HybridFileAppender*. El objetivo será mejorar el coste operacional de las dos operaciones que más latencia consumen en una gestión de log: escribir en fichero y almacenar los eventos en un buffer con su posterior extracción.

HybridFileAppender

Es la clase principal que extiende de *AppenderBase*, interfaz utilizada en Logback para poder capturar cada evento de log que se produce en la aplicación y gestionarlo. Esa operación es una de las que más latencia genera y va a ser el foco central a la hora de mejorar el rendimiento para tratar de reducir el coste total de la operación.

Para reducir la latencia, se han probado varias técnicas:

- Dividir el trabajo en varios procesos, de tal forma que hubiese uno de ellos que se encargase de la captura de los eventos y otro los recogiese. Otro proceso diferente sería el encargado de escribir los eventos recogidos a fichero [30]. Esta opción tenía demasiados inconvenientes: se creaban demasiados hilos y se sobrecargaba el uso de la memoria y los recursos del sistema. Por otro lado, no era capaz de reducir los términos de latencia que ofrecía Logback, por lo que se decidió no utilizar esta técnica.

- La siguiente técnica consistió en utilizar colas no bloqueantes (*Non-BlockingQueue*) para capturar los eventos de log de tal forma que los eventos se añadiesen en la cola de manera no bloqueante [31]. Si se producía una masiva llamada de eventos de log o se producían demasiadas peticiones para salir de la cola, se descartaba los eventos de log menos importantes y solo se mantenían los eventos de mayor prioridad.

  Para esto, se generó otro proceso que se encargaba de sacar los eventos de log de la cola de forma no bloqueante. Es decir, si la cola estaba vacía, se esperaba a que llegase algún evento de log. Si la cola estaba llena, se daba la opción de crear varios procesos que se dedicasen a sacar elementos de la cola, de manera que agilizase el procedimiento. Este proceso después era el encargado de enviar el evento de log a través del los procedimientos habituales para escribir el contenido de la información en un fichero.

  Se descartó esta técnica debido a la alta gestión de los dos procesos, la sincronización de todos los procesos conllevaba y a que se debía sacar los elementos en orden y por lo tanto añadían demasiada latencia. No obstante, con esta técnica se conseguía igualar e incluso superar en algunos casos a los ya implementadnos por Logback. Pero aún no era suficiente estable.

- La técnica que se ha implantado finalmente fue utilizar un buffer circular llamado *disruptor* [24] [25], donde se van almacenando todos los eventos generados y mediante otro proceso se van expulsando sin producir ningún tipo de bloqueo.

La técnica y uso del disruptor se empleará en el modo asíncrono ya que nos permitirá capturar un evento de log y almacenarlo en el buffer circular sin preocuparnos de cómo será gestionado el evento posteriormente y lo que es más importante, permitiendo almacenar más eventos que se vayan produciendo.

Para el caso del modo síncrono, el mecanismo a seguir será capturar los eventos y escribirlos directamente en fichero. Hasta que esta operación no finalice, se bloqueará el sistema y no dejará avanzar a la aplicación. La operación de escritura es por tanto la que se ha tenido que mejorar. La clase *HandleLogEvent* es la que se encargará de gestionar en qué momento hacer una u otra operación y la *WriteLogEvent* es común para los dos modos ya que es la que gestiona el nuevo método de escritura a fichero.

HybridFileAppenderParameters

Esta clase se utilizará para guardar todas las configuraciones, parámetros y valores por defecto que tenga el appender y todas aquellas que introduzca el usuario mediante el uso del fichero de configuración. De esta manera, cuando se produzca el primer evento de log en la aplicación, el sistema cargará todos los valores que se encuentren en esta clase y en el fichero de configuración. En caso de que el usuario no haya especificado los valores a través del fichero, se han establecido los parámetros con unos valores estándar por defecto.

HandleLogEvent

La clase HandleLogEvent administra y decide qué hacer con los eventos capturados en función de la configuración que se establece en el appender, bien sea el modo síncrono, asíncrono o mixto. Se han implementado dos tipos diferentes de manejos una vez capturado un evento de log:

- **Operación síncrona**: una vez capturado el evento, se envía inmediatamente sin pasar por el buffer circular y se envía al buffer de escritura a través de un canal donde posteriormente será escrito a fichero (manejado por el modo síncrono).
- **Operación asíncrona**: una vez capturado el evento, se almacena en el buffer circular. Otra operación se encarga de sacar los eventos del buffer circular de manera asíncrona (modo asíncrono y mixto) y posteriormente es enviado al buffer de escritura a través del canal para su posterior escritura a fichero.

WriteLogEvent

La clase *WriteLogEvent* se encarga de escribir en un fichero toda la información que está contenida en los eventos. Para mejorar la escritura en fichero, se probaron dos tipos de técnicas.

- La escritura en ficheros usando la clase *FileReader* [32]. Esta clase tiene métodos que nos permiten leer caracteres y escribirlos en memoria. El problema de estas clases es que no son lo suficientemente eficientes y no se consigue el rendimiento deseado, por lo que se descartó su uso.
- Utilizar *archivos mapeados en memoria (Memory Map Files)* [33]. Los elementos que son sacados del buffer circular (modo asíncrono) o los eventos que se capturan directamente sin utilizar buffer (modo síncrono), son directamente enviados a otro buffer para poder ser gestionados y elaborar su posterior escritura.

  Se investigó y se decidió utilizar esta técnica porque es muy útil en el espacio comercial de alta frecuencia, donde el sistema de comercio electrónico tiene que ser rápido y la latencia del conjunto de las operaciones ha de estar en el micro-segundo.

  Utilizar archivos mapeados en memoria tiene varias ventajas en:

- **Operatividad**. El sistema operativo se encarga de la lectura y la escritura e incluso si el programa se bloquea justo después de escribir en la memoria se encargará de escribir el contenido en un archivo.
- **Capacidad de almacenamiento**. La memoria utilizada para cargar el archivo de memoria asignada se encuentra fuera del espacio de almacenamiento dinámico de Java.
- **Compatibilidad**.
    - El lenguaje de programación Java es compatible con archivos de memoria asignada con el paquete java.nio y tiene *MappedByteBuffer* para leer y escribir en la memoria [34].
    - La mayoría de los principales sistemas operativos como Windows y UNIX soportan memoria mapeada y con arquitectura de 64 bits se puede asignar casi cualquier archivo en la memoria y acceder a él directamente utilizando el lenguaje de programación Java.
- **Rendimiento**. El acceso a los archivos asignados en memoria acceden:
    - Más rápidos que el acceso a los archivos a través de estándar normal de IO.
    - A más de un proceso y pueden actuar como memoria compartida con una latencia extremadamente baja.

Las desventajas y riesgos que se asumen utilizando archivos mapeados en memoria son:

- Aumento del número de fallos de página ya que el sistema operativo sólo carga una parte de archivo en la memoria. Si una página solicitada no se encuentra presente en la memoria, daría lugar a error de página.

Por lo tanto, con la nueva operación de escritura creada, se ha conseguido mejorar la tradicional escritura en fichero y de esta manera se ha conseguido optimizar el appender y por consiguiente, el sistema gestor de log.

4.2.3 Tiempos de de ejecución de las operaciones (latencias)

Las operaciones síncronas y asíncronas tienen diferentes latencias, es decir, cada una necesita un determinado tiempo para realizar una operación completa de capturar un evento de log entre el comienzo y el final de todo el proceso.

Operación síncrona

El tiempo de ejecución total de una operación síncrona consiste en sumar el tiempo que se emplea para capturar y recoger la información de un evento de log y escribirlo en fichero a través del buffer de fichero y el canal de escritura. La fórmula para calcular la latencia de una operación asíncrona es:

Latencia $_{total}$= $\mathbf{T}_{\textbf{calcular información}}$ + $T_{\text{escribir en fichero}}$

donde

- $\mathbf{T}_{\textbf{calcular información}} = T_{\text{mensaje de log}} + T_{\text{nivel de log}} + T_{\text{nombre del Hilo}} + T_{\text{Logger}} + T_{\text{Fecha y hora}} + T_{\text{Modo de registro de eventos}}$
    - $T_{\text{mensaje de log}} = T_{\text{nivel de log}} = T_{\text{nombre del Hilo}} = T_{\text{Logger}} = T_{\text{Modo de registro de eventos}} \approx 20$ nsg
    - $T_{\text{Fecha y hora}} \geq 100\text{nsg}$

- $T_{\textbf{escribir en fichero}} = T_{\text{acceso canal}} + T_{\text{acceso buffer}} + T_{\text{obtener permisos}} + T_{\text{escritura}} + (T_{\text{bloqueos}})^*$

El tiempo empleado en calcular la información de un evento de log es despreciable ya que es cuestión de pocos nanosegundos (aproximadamente 20nsg todas ellas) excepto el $T_{\text{Fecha y hora}}$ que necesita realizar llamadas al sistema para conocer el tiempo con precisión en milisegundos y eso requiere latencia.

(*) Para poder escribir el siguiente evento de log en el buffer, ha de terminar por completo una sola operación, teniendo en cuenta el tiempo de bloqueo que produce cuando llegan dos eventos de log consecutivos.

Operación Asíncrona

El tiempo de ejecución total de una operación asíncrona consiste en sumar el tiempo que se emplea para capturar y recoger la información de un evento de log y lo que tarda en meter el evento en el buffer circular. El resto de operaciones posteriores como la expulsión del evento de log para su escritura a fichero no afectan a la latencia de la operación ya que son manejadas asíncronamente por otro proceso lo que permite la entrada de nuevos eventos de log en el buffer circular incluso sin llegar a haber escrito en el fichero los eventos iniciales capturados.

La fórmula para calcular la latencia de una operación síncrona es:

Latencia $_{total}$= **T $_{\text{guardar en buffer circular}}$** + ($T_{bloqueos}$)*

donde:

- **T $_{\text{guardar en buffer circular}}$** = $T_{mensaje\ de\ log}$ + $T_{nivel\ de\ log}$ + $T_{nombre\ del\ Hilo}$+ T_{Logger}+ $T_{Fecha\ y\ hora}$+ $T_{Modo\ de\ registro\ de\ eventos}$
  - $T_{mensaje\ de\ log} = T_{nivel\ de\ log} = T_{nombre\ del\ Hilo} = T_{Logger} = T_{Modo\ de\ registro\ de\ eventos} \approx 20$ nsg
  - $T_{Fecha\ y\ hora} \geq 100$nsg

En este caso no se tiene en cuenta el tiempo que se tarda en escribir en fichero ya que no afecta a la latencia total de la operación, debido a que el buffer circular es asíncrono y permite que otro evento de log sea capturado consecutivamente.

(*) El tiempo de bloqueo del buffer solo existirá en situaciones de máxima saturación en las que recibimos una gran cantidad de eventos que el sistema no es capaz de gestionar y saturen el buffer circular. Para estas situaciones, se ha dotado la posibilidad de descartar los eventos en caso de saturación, con lo que nunca se llenaría el buffer y de esta manera se elimina el riesgo y ese tiempo puede ser tomado como 0.

Operación Mixta

Latencia $_{total}$= **T $_{\text{guardar en buffer evento}}$** + ($T_{extraer\ buffer\ evento}$ + $T_{escribir\ en\ fichero}$)

Dependiendo del nivel de log, se tratará como si fuese una operación síncrona o asíncrona y los tiempos serán o bien unos u otros.

4.2.4 Modos de funcionamiento

Modo Síncrono

El modo síncrono es responsable de recopilar los eventos de log producidos y escribirlos directamente a fichero. La operación de escritura en fichero es bloqueante, impidiendo capturar y avanzar a la aplicación hasta que la operación se haya completado.

Cualquier bloqueo supone un aumento de la latencia de una operación síncrona, que incluso puede ser muy alta si se recibe una llegada masiva de eventos de log.

Como la operación de escritura es costosa en latencia, se utilizará la técnica de escritura que se ha descrito en el apartado anterior. Los mapas de bytes buffer son capaces de reducir la latencia de una operación de escritura, pero existe el riesgo de que el buffer se llene, por lo que debe ser suficientemente grande para almacenar el mayor número de eventos de log que puedan generar en la aplicación.

El tamaño del buffer que se ha considerado suficiente para realizar las operaciones sin saturación ha sido de 512 bytes. Este valor se ha obtenido teniendo en cuenta que el tamaño de los mensajes que viajarán por el bus no serán mayores de 300 bytes y el tamaño de las cabeceras del los mensajes es de 20 bytes. Aún así, este valor se puede modificar en el archivo de configuración, dejando al usuario pleno control sobre este buffer si lo desea.

El buffer y el canal se obtuvieron utilizando la biblioteca *java.nio* [34] [35], que ofrece flexibilidad y sustituyen los *StringBuffers* para almacenar el contenido que posteriormente se almacenará, consiguiendo un aumento significativo de rendimiento.

Modo Asíncrono
El modo asíncrono recoge los eventos de log generados por la aplicación y los almacena en un buffer circular llamado disruptor. Esta operación permite introducir rápidamente eventos de log en el búfer y continuar con la captura de eventos de registro que puedan ocurrir después. Esta operación de inserción es no bloqueante y la ganancia en el rendimiento y disminución de latencia es muy significativa, siendo un [300% - 400%] mejor que el modo síncrono.

Si se produce una masiva afluencia de eventos de log, existe el riesgo de que el buffer pudiera llegar a saturarse. Esta condición se dará si la eliminación de eventos es más lenta que la inserción de elementos en el buffer. Para evitar esta situación, en el archivo de configuración se ha habilitado la activación del descarte (desactivado por defecto) que permite descartar un evento de log a su llegada sin tener que insertarlo en el búfer circular. Será el usuario de la aplicación quien decida si es necesario activar esta opción. La configuración se realizará mediante el fichero de configuración y a su vez requerirá especificar otras opciones:

- **Porcentaje de descarte**. Será una opción que indica cuándo se desea descartar los eventos de log en función de la capacidad que tenga el buffer. El valor predeterminado será descartar cuando el buffer esté a más del 85% de su capacidad.
- **Nivel de log de limpieza**. El usuario deberá elegir qué niveles de log quiere descartar. Se descartarán a partir del nivel de log que elija y todos aquellos niveles que sean inferiores al seleccionado.

Modo Mixto
El modo mixto recoge los eventos de log generados por la aplicación mediante las dos últimas técnicas y en función del nivel de los eventos de log, la operación que se realiza para capturarlos será síncrona o asíncrona.

Si los eventos de log son de nivel ERROR o WARN, entonces el appender utilizará la operación síncrona. No es la operación más eficiente en términos de rendimiento, pero es más segura y es la que menos recursos utiliza.

Este modo se añade para obtener unas buenas latencias de rendimiento sin verse afectado negativamente y en caso de bloqueo del buffer, el modo mixto se asegura de que los eventos de log de nivel más importante siempre se almacenarán en el fichero.

Los eventos de nivel de log TRACE, DEBUG e INFO son almacenados en fichero de manera asíncrona. Son los eventos de log de nivel menor de importancia y si el buffer se encuentra saturado, este modo considera que son los menos importantes y se podría perder alguno de los eventos de este nivel. La ventaja que tienen es que se almacenan muy rápido en fichero, por lo que no deberían tener problemas de bloqueo.

4.3 Appender Ultra Messaging: UMAppender

Una vez desarrollada la tecnología para poder capturar los eventos de log eficientemente (utilizando el buffer circular) y dar la posibilidad al usuario de utilizar eficientemente la escribir en fichero, se decidió utilizar estas funcionalidades e implementar un nuevo appender que pueda ser utilizado e integrado en el LLFB.

UMAppender (Ultra Messaging appender) almacena los eventos de log generados por una aplicación en un buffer circular, mientras otro proceso saca los eventos del buffer, serializa todos los datos de información y utiliza el bus de baja latencia (LLFB) para enviar los datos en forma de mensaje (cadena de bytes) por el bus.

Posteriormente, el appender dispondrá de otro proceso que será capaz de recibir los mensajes que son enviados por el bus de baja latencia y gestionarlos como el usuario quiera. La ventaja es que esta operación no supondrá ninguna latencia adicional *ya* que todo el proceso vuelve a ser asíncrono y permite la inserción de eventos de log en el buffer mientras que al mismo tiempo se está gestionando o enviando eventos por el bus.

UMAppender envía y recibe eventos de manera eficiente por el bus y explota todos los beneficios de manejar el LLFB:

- Evita tener que escribir en fichero los eventos de log mejorando el rendimiento de la aplicación.
- Utiliza canales diferentes de comunicación de otras aplicaciones que estén usando el bus para enviar y recibir mensajes. Es un proceso independiente que no depende de ningún otro dentro del LLFB. UMAppender se ha diseñado con un canal propio evitando interferir en otros procesos para no empeorar el rendimiento, tanto del mismo proceso como del resto de procesos.
- Cualquier aplicación que esté usando UMAppender puede gestionar los eventos de log que se produzcan en la aplicación por si misma (modo Local) o por cualquier otra aplicación que esté utilizando el LLFB.
- Cualquier aplicación que esté utilizando LLFB puede recibir y gestionar los eventos de log de otra aplicación conectándose al mismo canal de eventos de log de esa aplicación.

Las cualidades y capacidades que dispondrá este appender para cumplir con los requisitos inicialmente planteados son:

- Gestionará eficientemente los eventos de log capturados por aplicación suponiendo el menor coste operacional por evento de log producido.
- Utilizará el LLFB para que las aplicaciones puedan enviar y recibir los eventos de log utilizando este appender.
- Filtrará los eventos de log deseados por el usuario.
- Utilizará canales independientes a otras aplicaciones que usen el LLFB para enviar y recibir los eventos de log.
- Serializará los datos de una manera eficiente con la intención de reducir la congestión del bus y optimizar los envíos utilizando el máximo rendimiento del bus. Además, se podrá escoger los parámetros de información que se desean serializar y enviar por el bus.
- Evitará los casos extremos de máxima saturación y recogida de eventos descartando aquellos eventos que no sean necesarios o bien para los casos en los que el tamaño del buffer se ve saturado y necesita ser limpiado.
- Será compatible con las funcionalidades de Logback para que el usuario final pueda utilizar las funcionalidades que este ofrece a pesar de que no sea la mejor opción en lo que a rendimiento se refiere.
- Enviará los eventos de máxima prioridad de nivel ERROR y WAR directamente por el LLFB sin utilizar el buffer circular. Será una posibilidad que se le dará al usuario si desea que este tipo eventos de log siempre sean enviados y que nunca haya pérdidas por problemas de saturación.
- Utilizará un fichero de configuración para la puesta en marcha y selección de todos los parámetros por el usuario final.

4.3.1 Low Latency Framework Bus

Este appender utiliza el LLFB y a pesar de que su desarrollo no es parte de este proyecto de fin de carrera, se explicará brevemente su funcionamiento y las características más importantes.

El LLFB es una API de mensajería que tiene una envoltura de Ultra Messaging para mejorar su funcionalidad haciendo más transparente al usuario muchas de sus operaciones. LLFB también minimiza la latencia, maximiza el rendimiento y asegura la entrega de todos los mensajes en una amplia variedad de escenarios operacionales.

Esta API proporciona aplicaciones con funcionalidad de entrega de mensaje que añade un valor considerable para los servicios de red básicos. LLFB utiliza el modelo envío (fuente) / recepción (destino) de la mensajería, por el que uno o más programas receptor expresa su interés en un tema (topic) y uno o más programas envían mensajes a ese tema. Un tema puede ser visto como un flujo de datos que puede tener múltiples productores y consumidores.

Cualquier usuario que quiera utilizar el bus de baja latencia en el marco de su propia aplicación, primero debe crear un administrador (*Manager*), que es el objeto principal que se encargará de controlar todo lo que sucede en la aplicación. Sólo debe tener un administrador por cada aplicación y éste se puede configurar a través de los parámetros.

El administrador crea las conexiones necesarias, que es por donde viajan los mensajes. Para cada conexión hay un *Sender (fuente que envía mensajes)* responsable de enviar mensajes o peticiones de información y un Receiver (*receptor de mensajes*) que se encargar de recibir los mensajes y responder a las solicitudes. Las conexiones están interconectadas entre sí y se pueden configurar a través de sus parámetros. Todos estos conceptos se muestran esquematizados en la figura 10.

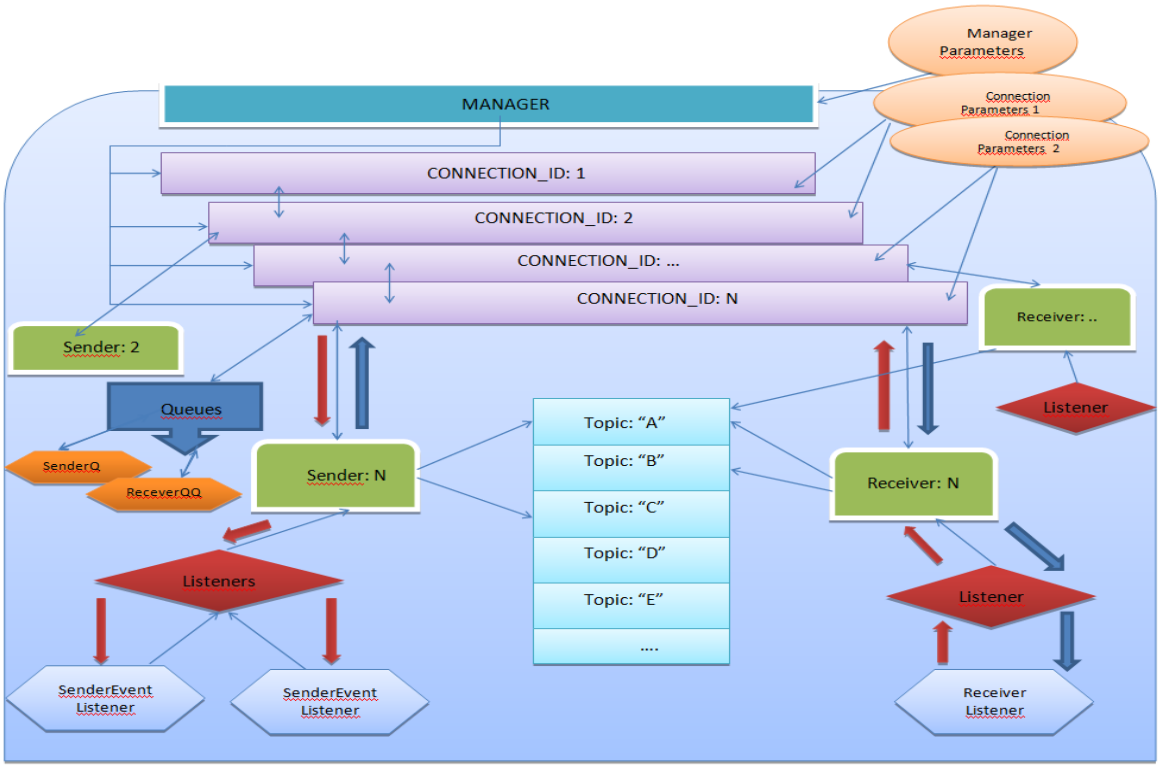


Figura 10. Esquema visual del Low latency Framework Bus.

Los emisores y los receptores están conectados a los topic para el intercambio correcto de los mensajes. Un remitente que envía un mensaje a través de un topic "A", será recibido por un receptor que está conectado a ese mismo topic. No es ningún impedimento que *el Sender* esté utilizando una conexión #1 y el receptor esté utilizando la conexión #2 para intercambiar los mensajes. El intercambio se produce por topic y no por número o tipo de conexión.
Los responsables de manejar los eventos (mensajes entrantes, solicitudes, etc.) que se producen en la escucha, son los *Listeners*. Estos son los que realmente recogen los mensajes y dan la posibilidad de manejarlos.

En caso de que haya una saturación de mensajes, cada conexión tiene una cola de eventos para almacenar los mensajes y liberar al remitente de la saturación de los receptores.

4.3.2 LogEventUM

LogEventUM es una clase especialmente diseñada para la creación de eventos de log cuando son capturados. Cuando una sentencia de log es capturada se genera una nueva instancia *LogEventUM* donde se asignan los parámetros de información en función del modo de log que esté seleccionado.

Esta clase en esencia es prácticamente igual que la clase LogEvent. La diferencia radica en que a esta clase se le han añadido dos métodos para serializar y des-serializar todo el contenido de los parámetros de información para ser enviados y recibidos por el bus.

LogEventUM implementa la interfaz *ILogEventUM* que ofrece una forma única para serializar la información. Además, al viajar los eventos de log por el bus, se han de añadir otros parámetros que identifican a los mensajes que viajan por el bus.

4.3.2.1 Serialización de datos

Aunque la serialización de datos no formaba parte de este proyecto, se estudió que se debía utilizar para enviar los eventos de log por el bus de baja latencia con la intención de mejorar el rendimiento del envío y recepción de los mensajes a través del bus.

Este mecanismo es usado para transmitir información a través de una red o en este caso, el bus de baja latencia para hacer persistente un objeto en un archivo o base de datos y distribuir la información de forma idéntica a varias aplicaciones o localizaciones y agilizar el transporte al máximo por el bus de baja latencia.

La serialización de datos que se ha implementado en este proyecto consiste en un proceso que codifica la información de los objetos *LogEventUM* que se encuentran en el buffer circular y que después se transmitirán a través del bus de baja latencia como una serie de bytes. Las series de bytes serán usadas para crear un nuevo objeto que es idéntico en todo al original, incluido su estado interno en la salida del bus.

Para serializar se han utilizado dos técnicas:

- **Protocol Buffers** [36]
  Es la primera técnica que es utilizó para serializar los datos. Protocol Buffers es un método para serializar los datos estructurados. Son una solución flexible, eficiente y automatizada para serializar datos estructurados. Como tales, son útiles en el desarrollo de programas para comunicarse entre sí a través de un bus o para almacenar datos.

  El propósito principal de Protocol Buffers es facilitar la comunicación de la red. Su simplicidad y velocidad hacen de Protocol Buffers una alternativa a los datos en los que la interoperabilidad con otros lenguajes o sistemas sean necesarios en el futuro.
  El método consiste en un lenguaje de descripción de la interfaz que describe la estructura de algunos de los datos y el programa se genera a partir del código fuente que puede estar en varios lenguajes de programación para generar o analizar una secuencia de bytes que representa los datos estructurados.

  Protocol Buffers ha sido desarrollado para uso interno, se ha hecho en los compiladores de Protocol para C + +, Java y está bajo un software libre.
  Protocol Buffers destacan por su sencillez y rendimiento. En particular, se ha diseñado para que sea más pequeño y más rápido que el XML.

  Protocol Buffers es muy similar al protocolo *Apache Thrift* [37] *utilizado por Facebook por ejemplo*, excepto que no incluye una pila de protocolo RPC concreta para utilizar los servicios definidos.

Para utilizar Protocol Buffers, se definen las estructuras de datos (los parámetros de información del appender) en un archivo de definición de proto (. Proto) y se compila con **protoc**. Esta compilación genera un código en el lenguaje deseado que puede ser invocado por las clases que se estén utilizando. En este caso, logEvent.proto producirá dos clases logEvent.pb.cc y logEvent.pb.h que definirán las clases Java para cada mensaje.

Protocol Buffers son serializados en un formato binario que es compacto, hacia delante-compatible, compatible con versiones anteriores, pero no auto descriptivo (es decir, no hay manera de saber los nombres, significado, o tipos de datos completos de los campos, sin disponer de una especificación externa.)

Como posteriormente se demostrará en el capítulo de pruebas, la utilización de ésta técnica no es eficiente ni consigue buenos resultados de rendimiento o latencia debido a la sobrecarga que utilizan sus librerías y la deficiencias introducidas por el generador de código que disponen para Java, por lo que se optó por implementar una técnica de serialización propia.

- **SerializaciónUM**
  Se decidió implementar una serialización de datos propia y hacerla sencilla y eficiente. Anteriormente, se estudió todas las características de Protocol Buffers y a pesar de que era una serialización eficiente, se pudo comprobar que añadía una pequeña carga de latencia a la hora de tener que ejecutar el generador de código interno y hacía que aumentase el tiempo más de lo deseado.

  Entonces, se creó e implantó una serialización propia. Para ello se analizaron todos los formatos de codificación de caracteres y cuáles iban a ser los formatos de serialización que se debía tener en cuenta [38]. Finalmente se desarrollo un serializador para el formato UTF-8, debido a que era el más estandarizado y el que se usaría en el resto de aplicaciones [39].

  El serializadorUM es el encargado de convertir a estructuras de bytes todos los tipos de datos existentes y enviar los bytes por el bus. En el otro extremo, cuando se recibiesen los bytes, se devolvería a su tipo de dato original. Solamente se ha configurado para el lenguaje Java y es utilizable para aplicaciones que utilicen el framework de baja latencia.

  El serializadorUM almacena todos los bytes generados por los tipos de datos en un buffer de bytes y utiliza un campo de desplazamiento (offset) para moverse por el buffer e identificar los bytes correspondientes a cada mensaje dentro del buffer.

4.3.3 Implementación

El UMAppender funciona de forma similar que el appender *hybridFileAppender*. El appender captura los eventos de log generados por una aplicación, recoge todos los parámetros de información en función del modo que esté seleccionado y guarda el evento en un buffer circular, permitiendo introducir un evento *logEventUM* en el buffer sin tener que esperar a que éste sea gestionado, por lo que esta operación es asíncrona y no bloqueante.

UMAppender

Esta clase es la encargada de recoger los eventos de log cada vez que se generan en la aplicación y de crear la instancia de *LogEventUM* con los parámetros de información en función del modo de captura de eventos que esté seleccionado en el fichero de configuración para guardar el evento de log en el buffer circular (disruptor).

Este appender se inicializa mediante el fichero de configuración y es ahí donde el usuario se encarga de configurar las diversas opciones que tiene disponible. Este appender solo almacena los eventos de log de forma asíncrona ya que es la operación que menos latencia genera y evita los posibles bloqueos.

No obstante, se le ha dotado de la posibilidad de efectuar el envío de los eventos de log sin utilizar el buffer circular, pero solo sería para modos extremos en los que el buffer esté saturado y se quiera salvaguardar los eventos de log de mayor nivel. Si se activa esta posibilidad en el fichero de configuración, los eventos de log de nivel TRACE y WARN serán enviados directamente por el bus sin pasar por el buffer. Esto siempre asegura el envío de los eventos más importantes, aunque esta operación es lenta e ineficiente para el sistema y podría ser bloqueante, pero fue necesario contemplarla por razones de diseño y se decidió añadirla.

UMAppenderParameters

Esta clase se utilizará para guardar todas las configuraciones, parámetros y valores por defecto que tenga el appender y todas aquellas que introduzca el usuario mediante el uso del fichero de configuración. De esta manera, cuando se produzca el primer evento de log en la aplicación, el sistema cargará todos los valores que se encuentren en esta clase y en el fichero de configuración. En caso de que el usuario no haya especificado los valores a través del fichero, el appender funcionará con los parámetros por defecto.

LogEventSender

La clase LogEventSender crea un proceso que se encarga de sacar los eventos de log del buffer. Una vez conseguida la instancia del evento, se serializa toda la información con la técnica anteriormente mencionada. Una vez serializado, se enviará la información a través del bus de baja latencia.

La primera vez que se envía un evento de log, se crea e inicializa una conexión con los parámetros por defecto en el bus de baja latencia y se conecta simultáneamente a los canales del bus. Se ha decidido crear unos canales independientes y sólo útiles para enviar y recibir los mensajes con los eventos de log que se produzcan en las aplicaciones que usen este appender para que no interfieran con los mensajes que se estén utilizando en otras aplicaciones que usen el LLFB y no perder rendimiento.

La operación de creación de conexión al bus y conexión a todos canales es lenta y la latencia se dispara en esta operación. Afortunadamente, sólo se produce una vez y el coste en latencia final es despreciable.

Los nombres de los canales por los que se va a enviar los eventos serializados están configurados de acuerdo con el nombre de la aplicación y al nivel de Log del evento. El nombre de los canales será siempre de la siguiente manera:

- Nombre de la aplicación que use el appender/ TRACE
- Nombre de la aplicación que use el appender / DEBUG
- Nombre de la aplicación que use el appender / INFO
- Nombre de la aplicación que use el appender / WARN
- Nombre de la aplicación que use el appender / ERROR

Dependiendo del nivel de log del evento que se desee enviar, se puede utilizar un canal u otro para recibir los mensajes. Esto permitirá más flexibilidad para seleccionar en la recepción que canal se desea escuchar y que niveles de log se van a recoger.

LogsEventReceiver

Esta clase crea un proceso que se conecta al bus y a los canales y escucha a todos los eventos del log que se envían por los canales, con la posibilidad de elegir a que canal conectarse.

Cuando LogsEventReceiver recibe un evento, lo recoge y utiliza los métodos para des-serializar todo el contenido que ha viajado por el bus. Posteriormente, en función del nivel de log, la interfaz *ILogEventListener* se encarga de manejarlos eventos con la todos los parámetros de información y el usuario será responsable de implementar y gestionar los eventos recibidos en cada caso. Realmente, esta clase es un Listener del framework de baja latencia.

Los pasos que sigue LogEventReceiver para hacer esta operación son:

- **Crear una nueva conexión para conectar al bus**. LogsEventReceiver crea una nueva conexión al bus de baja latencia. LogsEventReceiver tiene tres formas diferentes para conectarse al bus de baja latencia solamente diferenciados por los parámetros de conexión debido a que para conectarse al bus existen varias posibilidades.
- **Conexión al canal.** Después de conectarse al bus, el proceso ese conecta a un canal en particular. En este caso, se conectará a los canales utilizando el nombre de la aplicación que envía los mensajes y el nivel de log de los eventos. Se ha equipado la posibilidad de elegir por qué canal escuchar y dejar de escuchar ya que no siempre se querrá obtener todos los mensajes del mismo canal.
- **Recibir eventos de log por el canal.** El método del proceso que se encarga de recibir los mensajes que viajan por el bus y por cualquier canal es *onRcvNewMessage***.** Cada vez que recibe un mensaje por el bus, lo primero que se hará es des-serializar el mensaje. Una vez des-serializado, se creará la instancia de nuevo con todos los parámetros de información.
- **Administrar las nuevas instancias de evento.** El usuario puede gestionar eventos de log recibidas a través de la *ILogEventListener*. Esta interfaz recibe las nuevas instancias *logEventUM* des-serializadas. Una vez creada, se utiliza el método adecuado de que dispone la interfaz para controlar cada evento en función de su nivel de log. Por lo tanto, se han creado en la interfaz varios métodos (uno por cada nivel de log) para que el usuario pueda gestionar qué hacer con la instancia del evento.

  Ahí ya tendrá disponible toda la información para administrarla como desee. Puede escribir en fichero la información, mostrarla por una consola etc. La interfaz también recibe los mensajes del bus perdidos o erróneos.

4.3.4 Modos de uso

Modo con descarte

UMAppender implementa un búfer circular. En caso de una ocurrencia masiva de eventos de log, el buffer se podría saturar debido a que los eventos de log son insertados muy rápidamente en el buffer mientras que por el otro extremo son eliminados más lentamente, por lo que el buffer se llenaría.

Para evitar esta situación, en el archivo de configuración, se puede activar la opción “usar el modo de descarte” (por defecto desactivada), que permite descartar un evento de log a en su captura sin tener ser insertado en el búfer. Esta configuración exige, al igual que en el appender Híbrido, la configuración de dos parámetros en el fichero:

- **Porcentaje de descarte**. Será una opción que indica cuándo se desea descartar los eventos de log en función de la capacidad actual que tenga el buffer. El valor predeterminado será descartar cuando el buffer esté a más del 85% de capacidad.
- **Nivel de log de limpieza**. El usuario deberá elegir qué niveles de log quiere descartar. Se descartarán a partir del nivel de log que elija y todos aquellos niveles que sean inferiores al seleccionado.

Modo de envío directo

Como se ha indicado en la sección anterior, puede ocurrir una saturación de eventos de log en la entrada al buffer. Este modo lo que hace es facilitar el envío de los eventos de registro de nivel ERROR y WARN en el caso

de que el buffer se encuentre saturado, por lo que si la capacidad está casi llena, si este modo se activa por los parámetros de configuración del fichero, se envía los eventos de log directamente por bus de baja latencia.

Este modo está desactivado por defecto. Esta operación, hasta que no sé complementa el envío del evento por bus no devuelve el control a la appender, por lo que también es bloqueante y puede empeorar el rendimiento del sistema general, pero solo se activa si el buffer está bloqueado. Este modo disminuye el rendimiento del sistema, pero se ha buscado disminuir el tiempo de bloqueo del buffer y conseguir un equilibro mejorado entre los dos bloqueos insertados y recibir de forma segura los eventos más importantes de log.

Capítulo 5. Pruebas

Una vez finalizada toda la implementación, se mostrarán las pruebas con los resultados obtenidos por el nuevo sistema de gestión de log con todas las posibles configuraciones. Posteriormente, se hará una comparación de los resultados obtenidos por la nueva implementación y los ya obtenidos con los sistemas reales actuales para analizar si se ha mejorado o empeorado el rendimiento y comprobar si se ha conseguido cumplir los objetivos y requisitos planteados para cada una de las partes del sistema.

Las pruebas mostradas son los resultados finales del sistema del último ciclo de implementación. Es decir, son las pruebas realizadas en la última etapa de la producción final. Adicionalmente, en cada ciclo de producción, se realizaron pruebas para comprobar que se cumplían los objetivos y requisitos establecidos en cada ciclo y que las pequeñas modificaciones o nuevas insercciones de software no empeoraban el estado actual en cada fase.

Todas las pruebas han sido realizadas con las mismas condiciones. Se han realizado en el mismo computador con las mismas opciones de rendimiento. Es evidente que conseguir un estado ideal e idéntico en todas pruebas es imposible, bien por los procesos internos del sistema operativo o por las condiciones del computador, a si que se han tenido en cuenta todos los errores de medición introducidos a la hora de calcular los tiempos con la inteción de conseguir la máxima exactitud y realismo en todas las pruebas realizadas.

El equipo sobre el que se han realizado todas las pruebas ha sido:

- Dell – OptiPlex 790
- Intel® Core™ i5-2400 CPU @ 3.10Ghz
- 8,00 GB
- 64 bits OS Windows 7

Para conseguir resultados muy cercanos a la realidad, se realizaron todas las pruebas 5 veces, debido a que cada vez que se repetía una prueba bajo las mismas condiciones tóricas, los resultados variaban y en alguna ocasión considerablemente. De las 5 repeticiones de cada prueba se descartaron las que obtuvieron resultados extremos a la media de todas ellas.

Las pruebas han sido realizadas ejecutando pruebas de calentamiento previas a la prueba real para eliminar las latencias iniciales o cargas introducidas por el S.O. y la propia arquitectura del computador, como son los fallos compulsorios de memoria cache, fallos de paginación en memoria, o los tiempos iniciales introducidos por el propio sistema implementado como son los tiempos de carga de los managers o las conexiones al bus de baja latencia.

Todas estas operaciones iniciales de puesta en marcha generan una latencia muy superior al rendimiento que posteriormente ofrece el sistema, por lo que son eliminadas ya que solo son realizadas una vez en cada ejecución y no se ajustan al rendimiento real ofrecido por el sistema. Las latencias medias o máxima del sistema se verían afectadas en caso de mantener los tiempos iniciales.

5.1 Pruebas de Serialización

En el capítulo anterior, se habló sobre dos técnicas diferentes de serialización de datos. Una de ellas era Protocol Buffers y la otra técnica innovadora se llamó SerializaciónUM.

A continuación se muestra el estudio que se realizó para conocer cuál de las dos técnicas se debía utilizar para serializar los datos y utilizarla para enviar los datos por el LLFB.

5.1.1 Protocol Buffer vs SerializaciónUM

Test 1 – Test de Latencias y datos estadísticos.

El test consiste en enviar mensajes por el LLFB usando la capa de transporte TCP serializando los mensajes con las dos técnicas de serialización previamente vistas. Las características de estos test son:

- Red con un ancho de banda de 10 GB.
- Se enviaron por el LLFB 100.000 mensajes.
- Se enviaron 10.000 mensajes de calentamiento (para evitar latencias adicionales en el arranque de conexiones)
- Entre el envío de cada mensaje, hubo una espera activa de 10 milisegundos.

Se medirá el tiempo en enviar todos los mensajes serializados por el LLFB desde una fuente hacia un receptor en el bus. Se obtendrán las latencias, valores máximos y mínimos y los percentiles.

Los resultados se muestran conjuntamente en las figura 11 y 12 y las tablas 9 y 10.

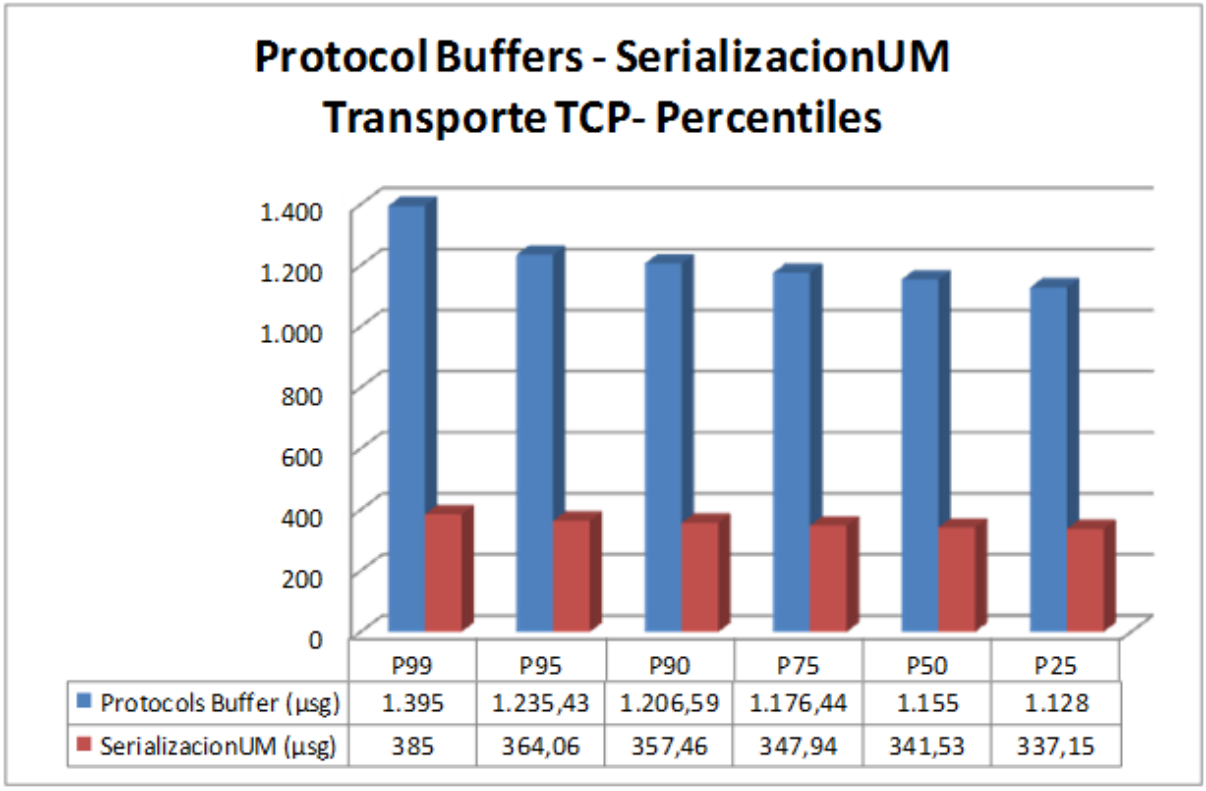


Figura 11. Vista de los percentiles enviando mensaje por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM.

Tabla 9. Resultados de los percentiles para Protocol Buffers y SerializaciónUM.

	Protocol Buffers	SerializacionUM (% mejor)
Percentil 99 (µsg)	1.395,142	72,41 %
Percentil 95 (µsg)	1.235,433	70,53 %
Percentil 90 (µsg)	1.206,599	70,37 %
Percentil 75 (µsg)	1.176,446	70,42 %
Percentil 50 (µsg)	1.154,542	70,41 %
Percentil 25 (µsg)	1.128,110	70,11 %

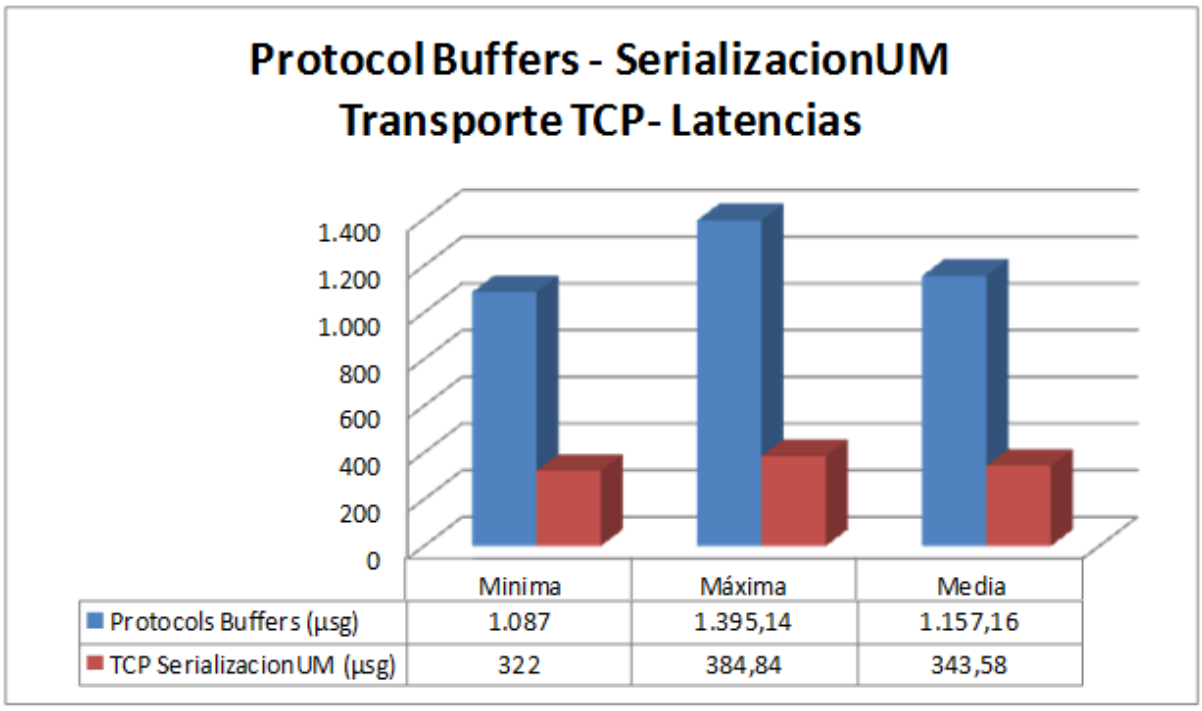


Figura 12. Vista de latencia *mínima, máxima y media* enviando mensaje por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM

	Protocol Buffers	SerializaciónUM	% mejor SerializaciónUM
Valor mínimo (µsg)	1.087.385	322.300	70,36 %
Valor máximo (µsg)	1.395.142,00	384.835,00	72,41 %
Latencia media (µsg)	1.157.164,00	343.584,00	70,30 %

Tabla 10. Resultados estadísticos en términos de latencia para la serialización de datos enviando mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM.

Test 2A – Test de Throughput para 10.000 mensajes.

El test consiste en enviar mensajes por el LLFB usando la capa de transporte TCP serializando los mensajes con las dos técnicas de serialización. Las características de los test son:

- Red con un ancho de banda de 10 GB.
- Se enviaron por el LLFB 10.000 mensajes.
- Se enviaron 1.000 mensajes de calentamiento (para evitar latencias adicionales en el arranque de conexiones)
- Entre el envío de cada mensaje, hubo una espera activa de 10 milisegundos.

Se medirá el tiempo en enviar todos los mensajes serializados por el LLFB desde una fuente hacia un receptor en el bus. Se obtendrá el Throughput, los mensajes por segundo y el nº de bytes que viajan.

Los resultados se muestran conjuntamente en la figura 13 y la tabla 11.

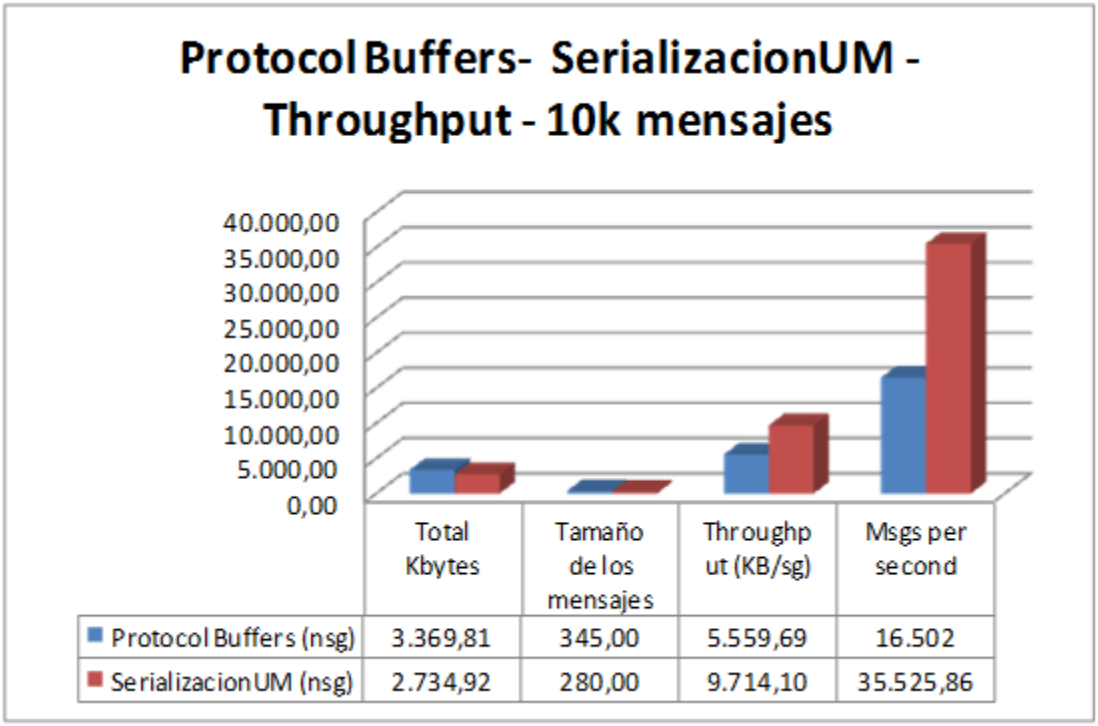


Figura 13. Vista del Throughput enviando 10.000 mensajes por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM.

Tabla11. Resultados estadísticos en términos de Throughput para la serialización de datos enviando 10.000 mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM.

	Protocol Buffers	SerializaciónUM	% mejor SerializaciónUM
Total Kbyte	3.369,81	2.734,92	18,84 %
Tamaño de los mensajes (Bytes)	345	280	18,84 %
Throughput (KB/sg)	5.559,69	9.714,10	74,72 %
Mensajes por segundo	16.502,43	35.525,86	115,28 %

Test 2B – Throughput para 100.000 mensajes.

El test consistió en enviar mensajes por el LLFB usando la capa de transporte TCP serializando los datos con las dos técnicas de serialización. Las características de los test son:

- Red con un ancho de banda de 10 GB.
- Se enviaron por el LLFB 100.000 mensajes.
- Se enviaron 10.000 mensajes de calentamiento (para evitar latencias adicionales en el arranque de conexiones)
- Entre el envío de cada mensaje, hubo una espera activa de 10 milisegundos.

Se medirá el tiempo en enviar todos los mensajes serializados por el LLFB desde una fuente hacia un receptor en el bus. Se obtendrá el Throughput, los mensajes por segundo y el nº de bytes que viajan.

Los resultados se muestran conjuntamente en la figura 14 y la tabla 12.

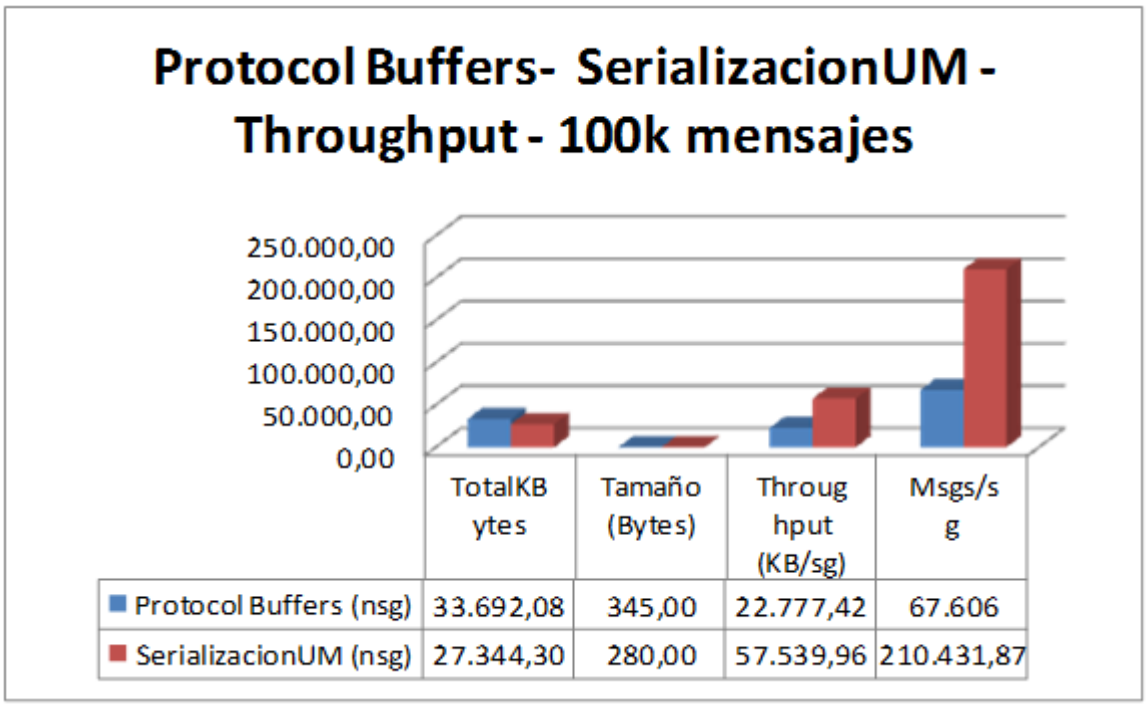


Figura 14. Vista del Throughput enviando 100.000 mensajes por el LLFB con modo de transporte TCP y usando serialización de datos Protocol Buffers y SerializaciónUM.

	Protocol Buffers (nsg)	SerializaciónUM	% mejor SerializaciónUM
Total Bytes	33.692,08	27.344,30	18,84 %
Tamaño de los mensajes (Bytes)	345	280	18,84 %
Throughput (KB/sg)	22.777,42	57.539,96	152,61 %
Mensajes por segundo	67.606	210.431,87	211,26 %

Tabla 12. Resultados estadísticos en términos de Throughput para la serialización de datos enviando 10.000 mensajes por el LLFB utilizando Protocol Buffers y SerializaciónUM.

5.1.2 Conclusiones

Como se ha podido observar en los test ejecutados, la utilización de la propia implementación mejora notablemente el rendimiento en cualquier situación sometida en contra de la que ofrece Protocol Buffers.

Los porcentajes de mejora has sido:

- Test de latencia. Se ha conseguido mejorar el rendimiento de la serialización un 70%.
- Test de Throughput. Se ha mejorado la serialización que va desde 18% y en algunos casos hasta el 211%.

5.2 Pruebas para los Appenders

Las pruebas para los appenders fueron realizadas bajo las mismas condiciones en todos los casos y fueron las mismas para todos los appenders.

Los pasos que se han seguido para realizar las pruebas han sido:

- Crear una aplicación de prueba que corra un bucle que ejecute 25.000 eventos de log, con una espera activa entre cada evento de 1 milisegundo para evitar el colapso del sistema. El funcionamiento será

como el que podría ser en una aplicación real, ya que se calcula que una aplicación necesita mandar eventos de log aproximadamente cada 1 milisegundo.
    - **HybridFileAppender** generó un fichero de salida que contiene 25.000 registros con toda la información de los parámetros de los eventos.
    - **UMAppender** envió por el bus 25.000 eventos de log y luego otro proceso muestra por pantalla el contenido de los parámetros de los eventos.
- Medir la latencia en ejecutar cada uno de los 25.000 eventos de log en cada caso.
- Obtener los datos estadísticos con esas 25.000 medidas. Los datos que se han obtenido han sido el percentil 99, 95, 90, la media, el valor máximo y el valor mínimo.
    - Antes de tomar las medidas de latencia, se hizo un calentamiento inicial con 100.000 eventos de log para iniciar las conexiones, rellenar los bloques de memoria vacíos y evitar todos los fallos del S.O o del propio sistema a la hora de iniciar conexiones.
- El tamaño de los buffers ha sido:
    - 8192 bytes para el buffer circular en todos los casos.
    - 521 bytes para el buffer de escritura a fichero.
- El nivel de los eventos de log ha sido: TRACE
    - Para el caso del modo de envío mixto en hybridFileAppender, se ejecutaron 2 niveles diferentes no consecutivos de log. El 50% de los eventos de log fue TRACE y el 50% fue ERROR. Esto se hizo con la intención de que el appender funcionase mezclando su forma de operar.

A continuación se mostrarán los resultados de las pruebas realizadas para los 2 appenders que se han implementado. Por cada appender, se han probado todos los modos de captura de los parámetros de los eventos (*only_message, Allcomplete y Logbacklayout*), para observar cuál es la diferencia entre cada uno de ellos. En el caso de HybridFileAppender, además se ha probado los 3 modos de escritura a fichero que dispone (*síncrono, asíncrono y mixto*).

5.2.1 HybridFileAppender

Los resultados obtenidos con el HybridFileAppender se muestran en la tabla 13.

HybridAppender

	Synchronous			Asynchronous			Mixed		
	Only_M	**AllComplete**	**Layout**	**Only_M**	**AllComplete**	**Layout**	**Only_M**	**AllComplete**	**Layout**
P99	17042	36876	29301	7823	9219	14247	8102	9219	13969
P95	10895	10962	13689	2794	3073	5587	3073	2794	5308
Avg	5543	6292	9219	2235	2235	3632	2235	2235	3802
Min	4146	5336	7263	1676	1676	2514	1676	1676	2514
Max	2679670	1341117	2747361	460952	1601432	3381943	667962	1686527	4934231

Tabla13. Resultados de todos los datos estadísticos en términos de latencia del HybridFileAppender. Todos los resultados que se muestran están expresados en nanosegundos.

5.2.2 UMAppender

Los resultados obtenidos con el UMAppender se muestran en la tabla 14.

UMAppender

	Only_M	AllComplete	Layout
Percentil 99	16873	9291	31203
Percentil 95	3631	3632	4470
Percentil 90	2235	2514	3632
Latencia media	2818	2911	3010
Mínimo	1676	1676	2514
Máximo	188666	4639668	4752978

Tabla14. Resultados de todos los datos estadísticos en términos de latencia del UMAppender. Todos los resultados que se muestran están expresados en nanosegundos.

5.2.3 Appender Logback

Con la misma aplicación creada para probar los appenders implementados, se ha realizado las mismas pruebas para los appender de Logback, para obtener unos resultados reales y en igualdad de condiciones, con la intención de poder hacer una comparativa totalmente realista.

Los resultados obtenidos se muestran en la tabla 15.

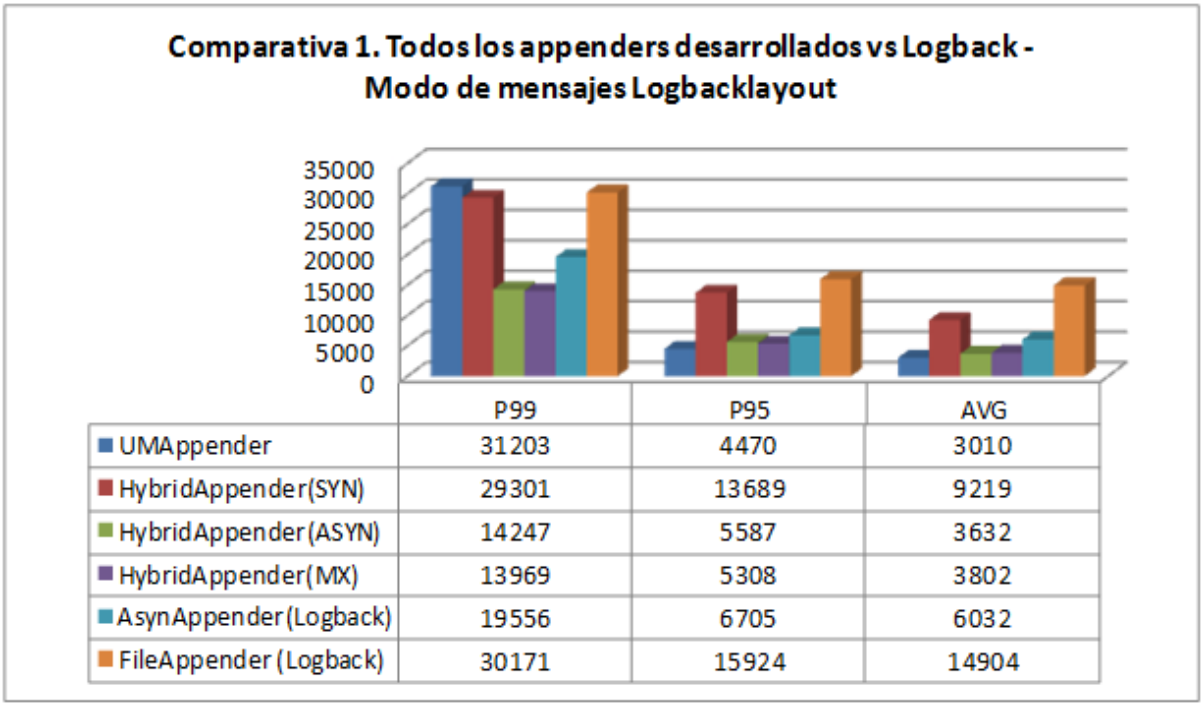
Appenders Logback

	FileReaderAppender	AsynAppender
Percentil 99	30171	19556
Percentil 95	15924	6705
Percentil 90	15454	6312
Latencia media	14904	6032
Mínimo	9499	2234
Máximo	2629664	890598

Tabla15. Resultados de todos los datos estadísticos en términos de latencia del FileReaderAppender de Logback y el AsynAppender. Todos los resultados que se muestran están expresados en nanosegundos.

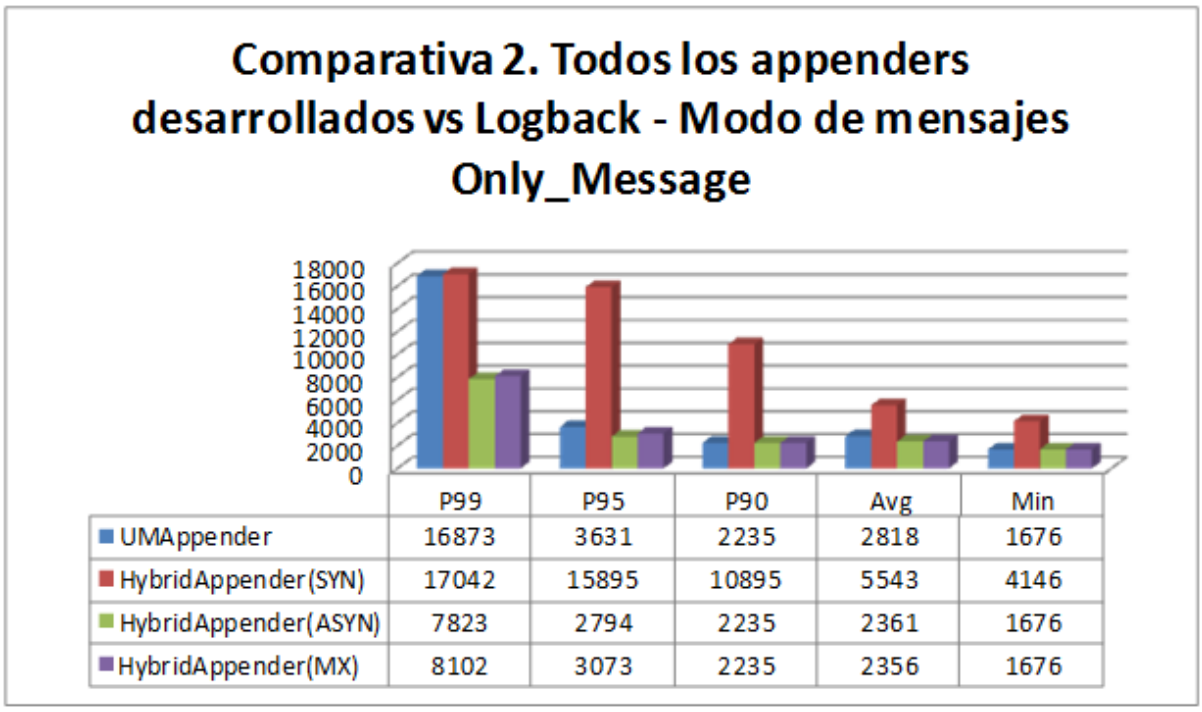
5.2.4 Comparativa de resultados

A continuación, se muestran las gráficas con todos los resultados obtenidos con el fin de poder apreciar la comparativa entre cada uno de ellos. Las figuras 13, 14, 15 y 16 muestran todos los datos.



	P99	P95	AVG
UMAppender	31203	4470	3010
HybridAppender(SYN)	29301	13689	9219
HybridAppender(ASYN)	14247	5587	3632
HybridAppender(MX)	13969	5308	3802
AsynAppender(Logback)	19556	6705	6032
FileAppender(Logback)	30171	15924	14904

Figura 15. Comparativa de los appenders usando el modo de captura del contenido del mensajes *LogbackLayout.*



	P99	P95	P90	Avg	Min
UMAppender	16873	3631	2235	2818	1676
HybridAppender(SYN)	17042	15895	10895	5543	4146
HybridAppender(ASYN)	7823	2794	2235	2361	1676
HybridAppender(MX)	8102	3073	2235	2356	1676

Figura 16. Comparativa de los appenders usando el modo de captura de contenido del mensaje *Only_Message.*

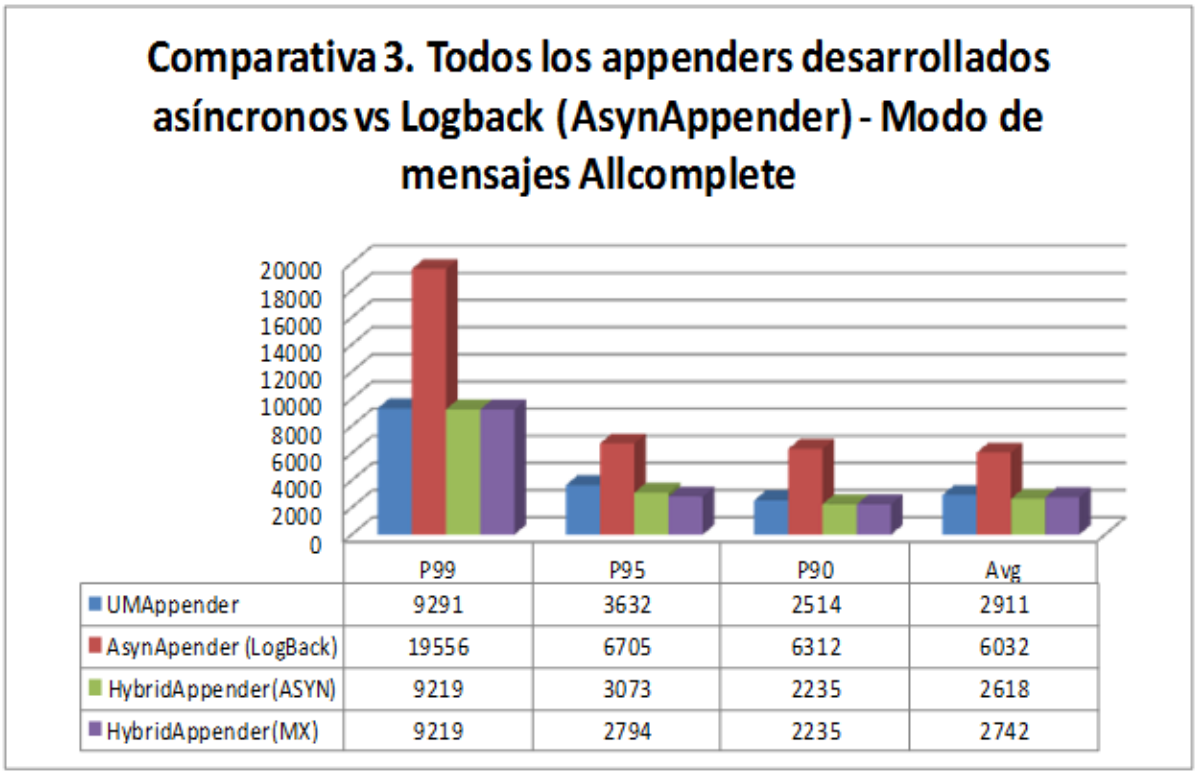


Figura 17. Comparativa de los Appenders usando el modo de captura de contenido del mensaje *AllComplete*

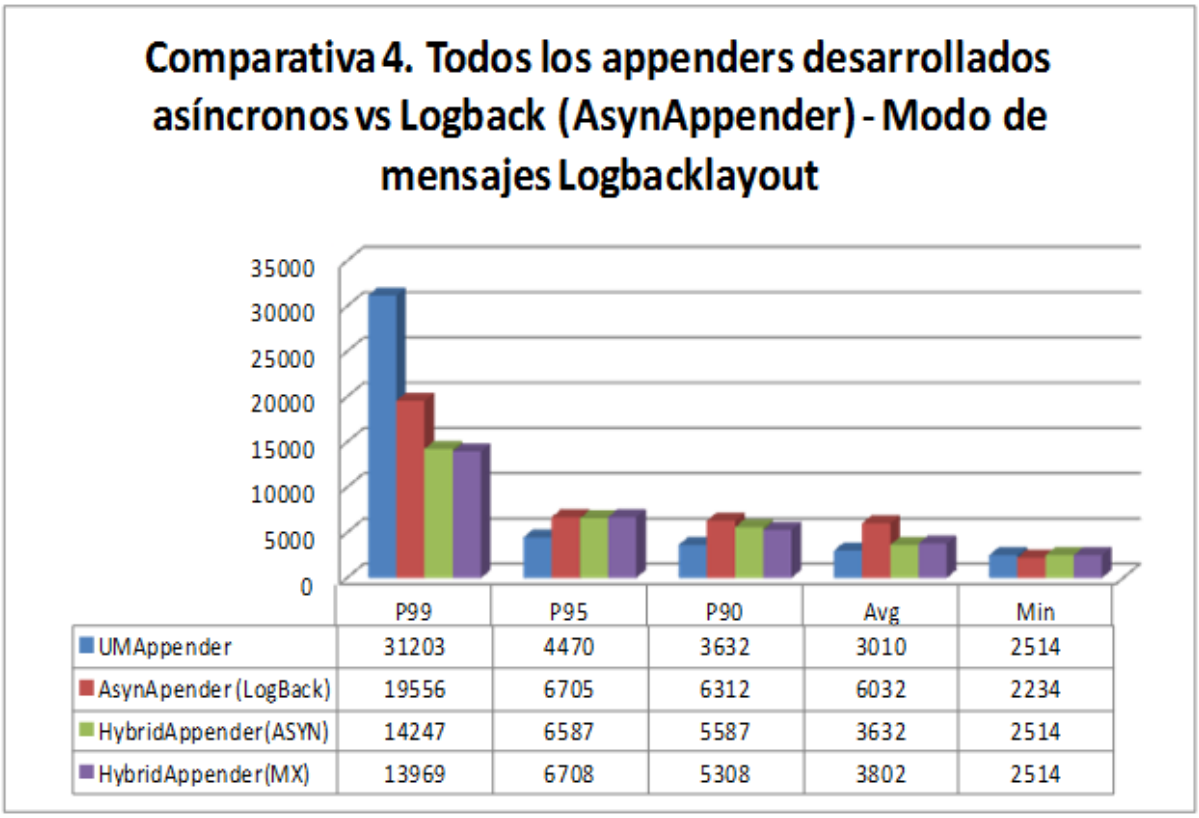


Figura 18. Comparativa de los Appenders desarrollados y el mejor appender de Logback (AsynAppender)

5.2.5Análisis de los resultados

UMAppender

- El appender cumple con los requisitos iniciales y está por debajo de los 5.000nsg de latencia media que exige un sistema de baja latencia.
- Los resultados de latencia han sido muy satisfactorios, especialmente en las mediciones para cualquier modo de envío. Se ha conseguido mejorar la latencia media respecto al appender asíncrono de Logback, logrando ser muy superiores (6.032nsg Logback vs 3.010nsg). Se ha mejorado el rendimiento en aproximadamente el 50% en términos de latencia para cualquier medición y se ha logrado rebajar el tiempo mínimo de latencia en ejecutar un evento de log. Además se puede observar que las latencias máximas no han empeorando.
- El modo más eficiente y más rápido, como cabía esperar, es Only_Message. Esto es debido a que el appender solo tiene que enviar por el bus los parámetros del contenido del mensaje y el nivel de log del evento.
- El modo de envío Allcomplete es un empeora un poco la latencia ya que este ha de calcular el tiempo exacto de ejecución del evento de log y le conlleva alguna llamada al sistema que añade latencia al sistema.
- Finalmente, utilizando el modo de envío LogbackLayout, el cual envía los eventos de log capturando todos los parámetros que hubiese capturado Logback, se observa que es también superior en rendimiento (excepto el P99) a usar el appender de Logback, por lo que podemos concluir que utilizar el bus de baja latencia para enviar los eventos de log, se consigue mejorar el rendimiento que al utilizar el sistema de Logback.

HybridAppender

- El appender cumple con los requisitos iniciales y está por debajo de los 5.000nsg de latencia media que exige un sistema de baja latencia. (excepto el modo síncrono, que a pesar de ser superior en rendimiento a de Logback, no consigue reducir la latencia establecida como objetivo.)
- El modo de escritura síncrono es mucho más lento que el modo de escritura asíncrono ya que como se ha comentado en capítulos anteriores, el síncrono es bloqueante y el asíncrono no es bloqueante, por lo que los tiempos de bloqueo, como se observa en los resultados, influyen notablemente en el rendimiento del sistema.
- El modo de escritura síncrono, a pesar de no lograr cumplir el objetivo, ha conseguido mejorar a los resultados conseguidos con el appender síncrono (FileReaderAppender) de Logback. (14.904nsg de Logback frente a los 9.219nsg)
- De los 3 modos de envió, Only_Message vuelve a ser el más eficiente (+35% superior en rendimiento) ya que solamente tiene que escribir a fichero el contenido del mensaje de los eventos de log.
- Utilizando cualquier modo de envío se consigue mejorar a los appenders de Logback. Esto se debe a que utilizando el mismo modo de escritura pero con las técnicas propias implementadas se consigue mejorar los términos de latencia, por lo que se deduce que se ha mejorado la implementación de Logback implementa.
- El modo asíncrono es el que mejores resultados obtiene, siendo muy superior en todas las mediciones tomadas respecto al appender asíncrono (AsynAppender) de Logback (6.032nsg frente a 3.632nsg). El modo de envío más eficiente es el Only_Message. Se consiguen unos resultados superiores en muchas medidas al 70%. El modo de escritura AllComplete, es ligeramente inferior al only_message, consiguiendo unos resultados en torno al 65%. Finalmente el modo LogbackLayout es el peor de los 3 modos de envío, pero en este caso sigue siendo superior a los resultados de Logback (15%).
- El modo de escritura asíncrono es muy superior al síncrono y al cualquiera de los modos implementado en Logback.
- El modo de envío mixto consigue unos resultados muy similares a los conseguidos por el appender asíncrono, (3.632nsg frente a los 3.802 del appender mixto). Aun así, es superior en todas las

mediciones respecto al Logback asíncrono, por lo que su utilización también puede resultar satisfactoria en cuanto a rendimiento.

- La diferencia respecto al asíncrono es que al utilizar la mitad de las veces la forma de enviar síncronamente, se asegura de que al menos siempre se escribirán los eventos ERROR y no se perderán ni se desordenarán. Se ha programado que solo utilice el envío síncrono para los eventos de log con nivel muy alto, que son los más importante de almacenar en fichero en caso de emergencia o deficiencia del sistema.

Capítulo 6. Integración en un software de mensajería instantánea: Trading Bank.

Como se ha visto en el capítulo anterior, los appenders implementados han logrado superar y mejorar el rendimiento del actual sistema que existe en el mercado para capturar y gestionar eventos de log y al mismo tiempo, se ha logrado integrar los appenders en un sistema de mensajería ultra rápida.

6.1 Integración del appender en el Low Latency Framework Bus

Comunytek ha desarrollado un sistema de mensajería de baja latencia para utilizarlo en las aplicaciones de Trading Bank. Estas aplicaciones son ultra rápidas. Su principal objetivo es ser capaces de enviar y gestionar el mayor número de mensajes por segundo.

El LLFB es el encargado de enviar y gestionar todo el envío y recepción de los mensajes. Actualmente, es capaz de enviar mensajes a más de 20 receptores que estén escuchando o esperando esos mensajes.

Todo el envío de mensajes, gestión, situaciones provocadas por todos los procesos y todo el trabajo que realiza el framework genera una serie de eventos. El encargado de gestionar, capturar y recoger todos los eventos es el sistema de gestión de log. El tema de este proyecto ha sido desarrollar ese sistema de log. A lo largo de esta memoria, se ha ido explicando cómo se ha desarrollado el sistema de gestión de log. Como ya se ha demostrado, se ha logrado implementar un sistema muy eficiente en rendimiento capaz de mejorar a los sistemas actuales del mercado.

Para que el sistema pudiera ser integrado y utilizado por el framework, se tuvieron que cumplir todos los requisitos inicialmente planteados, especialmente:

- El sistema de gestión de log debía ser de baja latencia y la gestión de un evento de log no podía suponer un tiempo mayor en latencia que el generado por el propio sistema de mensajes.
- El sistema de gestión de log no podía ser bloqueante, impidiendo ralentizar el trabajo que hace el framework y empeorar el rendimiento de todo el sistema.
- El sistema de gestión de log debía capturar todos los eventos sin pérdida de los mismos, aunque se debían insertar las distintas configuraciones opcionales para adaptarse a cualquier situación o aplicación.
- El sistema de gestión de log había de ser totalmente adaptable y modificable a los distintos cambios que se puedan realizar en el framework.

Una vez comprobado que se cumplieron los objetivos, se decidió integrar todo el código realizado dentro del LLFB y pasó a ser el motor y sistema principal de captura de eventos de log para las aplicaciones que lo utilicen. Ya que el sistema de log había sido diseñado para poder adaptarse al framework, su integración fue un paso sencillo de realizar.

Así pues, ahora el sistema de gestión de log es capaz de capturar todos los eventos de log generados por las aplicaciones que utilicen el LLFB e incluso puede ser configurado para que el framework recoja sus propias

sentencias de log que tiene internamente implementadas, de tal forma que se pueden capturar los eventos de una aplicación y los eventos que se genere el uso del LLFB, tanto por separado o al mismo tiempo.

6.2 Uso del LLFB y los Appenders: Aplicaciones de Trading Bank

El objetivo del LLFB es ser utilizado en aplicaciones que generen una cantidad elevada de mensajes y necesiten ser enviados y recibidos con la mayor rapidez posible. Este es el caso de las aplicaciones de trading. En esencia, un usuario que utiliza una aplicación de trading genera una determinada cantidad de órdenes (mensajes) y los envía a los mercados. Para poder enviar un solo mensaje, ha de pasar por un proceso intermedio de traducción, codificación, conexión a los distintos conectores de traducción, sistemas gestores de almacenaje y ordenación y decodificación hasta llegar a su destino, que normalmente sería un mercado. El mercado, a su vez, responde a los mensajes del usuario enviándole mensajes de respuesta y ha de seguir los mismos procesos hasta ser recibido. Es el mundo de la oferta y la demanda, lo que significa que el primero en ver la oferta es el primero en salir beneficiado.

El problema que tienen todo este tipo de aplicaciones es la cantidad de traducciones y codificaciones que hay en medio de la comunicación y el tiempo que conlleva realizar todo esto. Además, cada mercado utiliza un lenguaje diferente y un mensaje ha de ser codificado para todos estos mercados. Por lo tanto, es muy importante que todo este proceso se haga de manera eficiente, para que el usuario tenga las actualizaciones de sus mensajes y las ofertas de los mercados en el menor tiempo posible.

El tiempo es el factor clave de diferenciación en este tipo de aplicaciones. Una aplicación de cara al usuario que sea capaz de ofrecer toda la información 5 segundos antes que otra es claramente la que gana en el mercado. El hecho de ofrecer las ofertas y actualizaciones de los mercados antes que otra aplicación, supone la gran ventaja competitiva de poder comprar o ver las ofertas antes que el resto de usuarios, que en este caso son la competencia.

El LLFB está diseñado para trabajar en aplicaciones de este tipo. Se encarga de todo el envío, recepción, gestión, traducción, codificación y decodificación de los mensajes que son generados por el usuario y los que son enviados por los mercados. Es capaz de gestionar los dos lados de la conexión y elimina todo el proceso intermedio de traductores, conectores, bases de datos gestoras y demás que las presentes en las actuales aplicaciones.

El sistema de baja latencia se encarga de realizar todos los procesos, tanto el básico de envío y recepción de los mensajes, como el proceso intermedio que los otros sistemas realizan. Todo ello se hace de manera muy eficiente, rápida, generando que todo el cómputo de sus operaciones sea de latencia muy baja.

El sistema de gestión de log generado tiene como misión principal, dentro de una aplicación de trading, dar una explicación a las situaciones ocurridas. Genera un registro preciso de todos los movimientos sucedidos, tanto dentro de la aplicación como dentro del funcionamiento interno de todo el sistema. Otra de sus labores es ayudar a resolver cualquier tipo de error o problema que pueda ocurrir para ayudar a entender cómo fue el problema ocurrido.

En particular, los appenders implementados para este sistema de baja latencia también son capaces de realizar múltiples operaciones útiles dentro de una aplicación real de trading:

- Escribir en fichero (utilizando el HybridFileAppender) todo los eventos y movimientos que se hayan producido por parte del usuario. Es decir, si un usuario decidió enviar 200 mensajes y recibió 200 respuestas en un día y realizó varias operaciones de mercado (comprar acciones a un precio

determinado, vender futuros con una determinada cantidad a un precio determinado, etc., todos los detalles de todas las operaciones que realizó serán capturados.

Las aplicaciones de trading deben almacenar un histórico de todas las operaciones y movimientos realizados con todos los detalles, para que posteriormente puedan ser revisadas. Un usuario podría comprobar qué acciones realizo durante un día, semana, mes, a lo largo de un año. El sistema capaz de generar toda esa información sería el appender HybridFileAppender. Todo ello sería generado sin suponer una pérdida de rendimiento en la aplicación. Además se pondrá a disposición del usuario la configuración de todo el appender para elegir las opciones de captura de los eventos.

- Todas las operaciones internas que realiza la aplicación (enviar el mensaje 0xF3249, traducción, errores del sistema, etc.) van a quedar registradas en un fichero. Para ello se utilizaría el appender HybridFileAppender, que es el encargado de realizar esta operación de manera eficiente.

- Todos los eventos ocurridos, tanto los movimientos que realiza un usuario que maneja la aplicación como las acciones que realiza el propio sistema de baja latencia, pueden ser enviados a otro sistema de baja latencia, a otro usuario o a un mercado para analizar los datos y realizar algún tipo de operación con ellos. Esta tarea sería realizada por el *UMAppender*, que se encargaría de enviar (al mismo tiempo que el sistema esté realizando otras tareas o incluso utilizando otros appenders) todos los eventos de log producidos a un receptor y por un canal apropiado. Todo esto se realizaría de manera eficiente y sin suponer una carga adicional al sistema. Ésa es una de las razones por las que se decidió implementar este tipo de appender.

El funcionamiento de los appenders es invisible al usuario, que solo vería una interfaz gráfica para poder manejar toda la aplicación sin conocer como está realmente implementado

Los sistemas de gestión de log son muy importantes en aplicaciones de este tipo, debido a que se generan múltiples cantidades de mensajes y operaciones y muchas veces convienen ser revisadas. El registro de los eventos de log puede ayudar a resolver en muchos casos los detalles de estas operaciones realizadas cuando no estén claras o cuando se necesite resolver algún tipo de conflicto.

Capítulo 7. Conclusiones.

Del trabajo realizado en este proyecto se obtienen una serie de conclusiones respecto a la utilización de un sistema de log integrado en un sistema de mensajería de baja latencia.

La conclusión más importante es que nuestra solución ha conseguido cumplir los objetivos y requisitos en cuanto a latencia, rendimiento, flexibilidad, y capacidad. Esta conclusión se puede obtener a raíz de los resultados obtenidos, puesto que el rendimiento que nos muestra es alto y comparado con otros sistemas es muy superior.

Una ventaja de nuestro sistema respecto a otros sistemas tiene que ver con el rendimiento y latencia, puesto que en las pruebas realizadas se consigue mejorar en casi todos los casos todos los aspectos de latencia, consiguiendo en algunos casos mejoras de más de un 70%.

El sistema soporta final implementado está preparado para soportar situaciones extremas, como puede ser una llegada masiva de mensajes y eventos de log que sobrecarguen el sistema. Además, es capaz de controlar este tipo de situaciones sin verse afectado el rendimiento y por otra parte puede realizar la gestión selectiva de eventos de log para ese tipo de situaciones, descartando aquellos que carecen de importancia y así tratando de liberar la carga de los dispositivos.

También hay que tener en cuenta que nuestro sistema es muy sencillo de instalar, configurar y usar. Esta valoración es ciertamente subjetiva, ya que no se puede medir con un test, sino que es cuestión de la observación por parte del usuario al manejar el sistema respecto de sus alternativas. Toda la configuración del sistema se puede realizar mediante sencillos ficheros de configuración, donde tan solo se deben aportar las opciones deseadas para ejecutar el sistema, sin la necesidad de conocer el funcionamiento interno del sistema.

Al tener un importante rendimiento, se puede concluir que nuestro sistema es viable para la puesta en un sistema en producción. También cabe considerar la facilidad de integración en el sistema de mensajería de baja latencia, ya que ha sido diseñado e implementado con ese objetivo.

Por tanto después de todo el trabajo realizado en el sistema de gestión de log, se puede concluir que resulta muy útil para sistemas que requieren baja latencia y generen una gran cantidad de mensajes por segundo y que deseen almacenar todo el contenido de los mensajes en un fichero.

Capítulo 8. Líneas futuras de trabajo

Se ha implementado un sistema de gestión de log que ha sido integrado en un sistema de mensajería de baja latencia. A día de hoy, todavía no ha sido desarrollado ni implementado para ninguna aplicación, pero no obstante ha sido pensado para ser utilizado como aplicación de trading.

Las líneas iniciales de futuro a muy corto plazo es continuar desarrollando todas las partes que conforman el sistema de mensajería de baja latencia para poder ser utilizado como aplicación de trading. A día de hoy no está finalizado y aún queda por desarrollar partes importantes como son los sistemas gestores de clearing y quoting, partes específicas de un sistema de trading.

Una vez finalizado todo el proceso, habría que crear una interfaz gráfica y crear toda la documentación necesaria para su correcta utilización.

En cuanto al sistema de gestión de log, se podría seguir trabajando en el si se demuestra que, a pesar de su gran rendimiento, no es suficiente para hacer todas las operaciones del sistema lo suficientemente rápido para determinadas situaciones. Para sacar esta conclusión, sería necesario probarlo en una aplicación real y estudiar cuál es su comportamiento, ya que las pruebas realizadas han sido en pequeñas aplicaciones y al añadir nuevos factores se podría descubrir algún error que necesite ser reparado.

Las líneas de trabajo futuro sobre las que se podría trabajar en el sistema gestor de log serían:

- Reducir la latencia. El objetivo ideal sería conseguir capturar todos los eventos de log generados por la aplicación costando la menor latencia. Todo sistema que se acerque a 0 de tiempo de latencia será un sistema ideal. Así pues, se podría intentar reducir el tiempo de todas las operaciones (aún mas) hasta acercarnos lo máximo posible a este valor.

- Investigar y mejorar cómo conseguir capturar el parámetro de información que captura el tiempo exacto cada vez que se ejecuta un log sin generar tanta latencia. Esta operación, como se ha demostrado en los capítulos anteriores, es la que más coste operacional supone. La idea sería buscar un mecanismo que nos permita capturar este tiempo sin añadir tanta latencia.

- Implementar un sistema propio de formato de mensajes: actualmente los appenders utilizan el que viene implementado en Logback, ya que en un primer momento no se consideró necesario. Podría darse el caso de que más adelante se necesite implementar uno propio con la posibilidad de mejorar el rendimiento que ofrece el de Logback, ya que como se ha visto, utilizar ese sistema empeora el rendimiento del appender.

- Incorporar nuevas opciones configurables para los ficheros de configuración o para alguno de los appenders implementados. Quizás en un futuro se necesite realizar operaciones más concretas, con más detalles u opciones a la hora de configurar los appenders, como pueden ser opciones programables de tiempo de guardado, momentos para almacenar los eventos etc.

- Añadir al sistema gestor de log nuevos appenders. Por ejemplo, se podría desarrollar un appender que sea capaz de mostrar los eventos de log por pantalla, ya que los actuales generan demasiada latencia y son ineficientes para un sistema como el que está pensado.

Bibliografía

1. **Comunyteck.** "Solutions for mission critical enviroments: P'nT Trading". [Online] 2013. [Visitado: 06 01, 2012.] http://www.comunytek.com/en/solutions/plugntrade/57-pnt-trading.

2. **FiMarkets.** "Front, middle and back-office". [Online] 2013. [Visitado: Agosto 6, 2012.] http://www.fimarkets.com/pagesen/back-middle-front-office.php

3. **QOS.ch.** "Logback Proyect". [Online] 2012. [Visitado: 8 10, 2012.] http://logback.qos.ch/.

4. **Apache Software Foundation.** "Loggin services: Apache log4j 1.2". [Online] 2.0, 2012. [Visitado: Agosto 10, 2012.] http://logging.apache.org/log4j/1.2/.

5. **QOS.ch.** "SLF4J: Simple Logging facade for Java". [Online] http://www.slf4j.org/, 2012. [Visitado: Agosto 9, 2012.]

6. **Informatica Corporation.** "Informática: Ultra Messaging". [Online] 2012. [Visitado: Agosto 9, 2012.] http://www.informatica.com/es/products/messaging/.

7. **The Apache Software Foundation.** "Apache logging services". [Online] 2.0, 2012. [Visitado: Agosto 11, 2012.] http://logging.apache.org/.

8. **Ceki Gülcü, Scott Stark.** *"The complete log4j manual".* San Francisco : QOS.ch, 2010.

9. **QOS.ch.** "Logback: Reasons to prefer logback over log4j. [Online] 2012. [Visitado: Agosto 9, 2012.] http://logback.qos.ch/reasonsToSwitch.html.

10. **Courtine, Benoit.** "Logback ou Log4J". [Online] Septiembre 13, 2010. [Visitado: Agosto 13, 2012.] http://blog.courtine.org/2010/09/13/logback-ou-log4j/.

11. **SpringSource and the Groovy Community.** "Groovy: A dynamic languaje for the Java platform". [Online] 2013. [Visitado: Agosto 16, 2012.] http://groovy.codehaus.org/.

12. **Dierk König, Guillaume Laforge, Paul King, Jon Skeet, and Hamlet D'Arcy.** *Groovy in Action.* s.l. : Softbound print, 2009. Vol. Second Edition.

13. **Ceki Gülcü, Sébastien Pennec.** "Logback: Appenders". [Online] QOS.ch, 2012. [Visitado: Agosto 5, 2012.] http://logback.qos.ch/manual/appenders.html.

14. **B.W. Boehm.** A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61-72, 1988.

15. **Sommerville, Ian.** *Ingeniería del software.* Madrid : Pearson Educacion S.A, 2005. 7ª Edición.

16. **B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea.** *Java Concurrency in Practise.* s.l. : Addison-Wesley Professional, 2006.

17. **J.Bloch.** *"Effective Java".* s.l. : Addison-Wesley, 2008. 2nd ed.

18. **D. Lea.** "Concurrent Programming in Java: Design Principles and Patterns" 2nd ed. Addison-Wesley Professional. 1999

19. **Oracle.** "Interface Lock: JavaTM 2 Platform". [Online] 2010. [Visitado: Septiembre 29, 2012.] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Lock.html.

20. **ej-technologies GmbH.** JProfiler. [Online] 2013. [Visitado: Octubre 25, 2012.] http://www.ej-technologies.com/products/jprofiler/overview.html.

21. **G. Amdahl**, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967

22. **T. H. Cormen**, C. E. Leiserson, R. L. Rivest, C. Stein "Introduction to Algorithms", 3rd ed. MIT Press. 2012.

23. **Oracle.** "Java 2 SDK SE Developer Documentation". [Online] 2010. [Visitado: Septiembre 20, 2012.] http://docs.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html.

24. **LMAX-Exchange.** "LMAX Disruptor: High Performance Inter-Hilo Messaging Library". [Online] Noviembre 13, 2012. [Visitado: Octubre 1, 2012.] http://lmax-exchange.github.com/disruptor/.

25. **M. Thompson, D. Farley, M. Barker, P. Gee, A. Stewart**. *Disruptor:* High performance alternative to bounded queues for exchanging data between concurrent hilos. LMAX limited. 2011.

26. **Torrijos, Ricard Lou.** Programacion en castellano: Recolector de basura de Java. [Online] 2011. [Visitado: Septiembre 23, 2012.] http://www.programacion.com/articulo/tutorjava_nivel_basico_97/9.

27. —. "Interface Condition: JavaTM 2 Platform". [Online] 2010. [Visitado: Septiembre 29, 2012.] http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html.

28. **Vogel, Lars.** Java Concurrency (Multihiloing). [Online] Vogella.com, Enero 18, 2013. [Visitado: Septiembre 15, 2012.] http://www.vogella.com/articles/JavaConcurrency/article.html#hilopools.

29. **Ceki Gülcü, Sébastien Pennec.** "Logback: Layouts". [Online] QOS.ch, 2012. [Visitado: Septiembre 13, 2012.] http://logback.qos.ch/manual/layouts.html.

30. **Oracle.** "Java 2 SDK SE Developer Documentation". [Online] 2010. [Visitado: Agosto 12, 2012.]

31. **Oracle.** "JDKTM 5.0 Documentation" . [Online] 2010. [Visitado: Septiembre 17, 2012.] http://docs.oracle.com/javase/1.5.0/docs/index.html.

32. **Oracle —**. FileReader: JavaTM 2 Platform. [Online] 2010. [Visitado: Octubre 6, 2012.] http://docs.oracle.com/javase/1.4.2/docs/api/java/io/FileReader.html.

33. **Baldwin, Richard G.** Introduction to Memory-Mapped IO in Java. [Online] Developers.com, Noviembre 26, 2002. [Visitado: Octubre 3, 2010.] http://www.developer.com/java/other/article.php/1548681/Introduction-to-Memory-Mapped-IO-in-Java.htm.

34. **Pontarelli, Brian.** "High-Performance I/O with Java NIO". [Online] Septiembre 1, 2005. [Visitado: Octubre 9, 2012.] http://www.drdobbs.com/jvm/high-performance-io-with-java-nio/184406242.

35. **Oracle.** "Java NIO: JavaTM 2 Platform". [Online] 2010. [Visitado: Octubre 9, 2012.] http://docs.oracle.com/javase/1.4.2/docs/api/java/nio/package-summary.html.

36. **Developers, Google.** Developer Guide: Protocol buffers. [Online] Abril 2, 2012. [Visitado: Octubre 16, 2012.] https://developers.google.com/protocol-buffers/docs/overview.

37. **Apache Software Foundation.** Apache Thrift. [Online] 2012. [Visitado: Octubre 16, 2012.] http://thrift.apache.org/.

38. **(eds), J.D. Allen et al.** "The Unicode Standar Version 6.2 - Core Specification". [Online] Unicode Consortium, 2012. [Visitado: Noviembre 3, 2012.] http://www.unicode.org/versions/Unicode6.2.0/.

39. **F.Yergeau.** "UTF-8, a transforamtion format of ISO 10646". [Online] 2003. [Visitado: Noviembre 4, 2012.] The Internet Society , 2012- http://tools.ietf.org/pdf/rfc3629.pdf.

Anexo: Contenido del CD

El contenido del CD consta de ésta memoria en formato pdf y un documento técnico titulado "*Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*", que explica el desarrollo e implementación interna de esta novedosas estructura de datos, cuya utilización en éste proyecto ha sido clave para lograr los objetivos.